ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Reinforcement Learning in problemi di controllo del bilanciamento

Tesi in
MACHINE LEARNING

Relatore:
Prof. Davide Maltoni

Presentata da:
Michele Buzzoni

III sessione di laurea
Anno Accademico 2016 - 2017

Introduzione

Il reinforcement learning è un approccio di machine learning (e più in generale di Intelligenza Artificiale) esploso negli ultimi anni che sta avendo notevole successo in molti settori applicativi sia a livello concreto/pratico che in ricerca e letteratura. Si pone come obiettivo della tesi lo studio di algoritmi di reinforcement learning capaci di istruire un agente ad interagire correttamente con gli ambienti proposti con lo scopo di risolvere i problemi presentati. Nello specifico i problemi verteranno su un argomento comune: il balancing, ovvero problemi legati all'equilibrio. In particolare vengono presentati tre ambienti per il learning: due sono legati al conosciuto "cart-pole problem" in cui l'ambiente è composto da un carrello su cui è posto un palo. L'agente, muovendo il carrello, dovrà mantenere bilanciato il palo impedendo la sua caduta. Questo problema è realizzato in due varianti: una variante semplice in cui il carrello è legato ad un binario e quindi i suoi movimenti sono solo due (avanti, indietro), mentre la seconda variante prevede un ambiente più complesso in cui il carrello è slegato dai vincoli del binario e può quindi muoversi in 4 direzioni diverse. L'ultimo ambiente consiste di un piano quadrato su cui è posta una pallina. Il compito dell'agente è quello di mantenere la pallina sul piano, imparando a muovere opportunamente il piano stesso. Anche questo problema viene trattato in due varianti, una semplice ed una complessa, ma l'ambiente realizzato è il medesimo. Questa tesi presenta quindi due algoritmi per risolvere i problemi appena elencati: un algoritmo di Q-learning con uso di una Q-table per la memorizzazione delle componenti stato-azione e uno di Q-network in cui la Q-table viene sostituita da una rete neurale. Gli ambienti legati ai problemi che verranno affrontati sono realizzati attraverso pyBullet, libreria per la simulazione 3D di corpi solidi che viene integrata con Gym openAI, toolkit per la programmazione in ambito machine learning che offre semplici interfacce per la costruzione di nuovi ambienti. Prima di parlare degli ambienti realizzati, delle tecnologie messe in campo, e degli algoritmi sviluppati, la tesi parte con un capitolo introduttivo dei concetti di base e delle meccaniche necessarie allo sviluppo degli algoritmi proposti per poi discutere dei

risultati ottenuti nei tre ambienti e comparati tra loro.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Per chiarire cosa significa utilizzare algoritmi di Reinforcement Learning è bene partire dai concetti base da cui è composto. Pertanto in questo capitolo introduttivo viene definito il RL in termini delle sue componenti quali Policy, funzione valore, condizione di ottimalità di Bellman e in termini di controllo ottimo di un processo decisionale di Markov.
Il capitolo comprende anche l'analisi di alcuni tra i più comuni algoritmi come SARSA e Q-Learning per poi spostarsi su algoritmi più avanzati basati su reti neurali come Deep Q-Network. La trattazione trae spunto da [1].

1.1 Reinforcement learning

Il reinforcement learning, considerato una branca del machine learning, permette ad un agente di scoprire in totale autonomia il corretto comportamento da eseguire. L'apprendimento dell'agente è dato dalla continua interazione con l'ambiente da cui ottiene uno stato, e dalle azioni che l'agente stesso intraprende e che vanno a modificare lo stato stesso. L'ambiente è definito come tutto ciò che circonda l'agente e con cui esso può interagire.
In un dato time step t, sia l'agente che l'ambiente si trovano in uno stato s, che contiene tutte le informazioni rilevanti sulla situazione, come ad esempio la posizione di un oggetto. Dallo stato s può essere eseguita un'azione a. Otteniamo quindi una coppia stato-azione che può far parte sia di un insieme discreto di coppie stato-azione, sia continuo. Prima di passare al prossimo time step $t+1$, l'agente riceve un premio (reward) che viene trasferito al prossimo stato. La policy permette di determinare quale azione scegliere in un determinato stato. Le policy possono

essere deterministiche, quando la stessa azione viene intrapresa per un dato stato, oppure probabilistica, quando l'azione viene scelta in base ad una qualche calcolo di distribuzione tra le azioni e lo stato dato. Il funzionamento di algoritmi di reinforcement learning può essere sintetizzato nella figura 1.1.

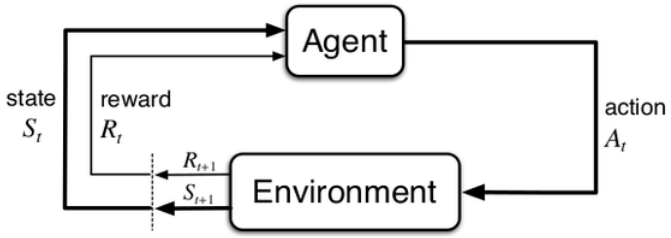


Figura 1.1: Funzionamento Reinforcement learning

Il reinforcement learning è "imparare cosa fare", come mappare le situazioni in azioni, così da massimizzare il reward di ritorno dato dall'esecuzione dello step. All'agente non viene detto quale azione intraprendere, deve scoprire invece quale azione porterà ad ottenere un reward maggiore semplicemente provando ed esplorando l'insieme delle azioni possibili. Nei problemi più interessanti le azioni possono condizionare non solo il reward immediato ma anche la successiva situazione e stato che si verrà a creare, e quindi condizionare anche i successivi reward. È importante fare una distinzione tra i vari algoritmi di learning esistenti e algoritmi di reinforcement learning trattati in questa tesi. Il reinforcement learning è diverso dal **learning supervisionato**, in cui si parte da un training set di esempi già classificati che quindi offrono all'algoritmo già un certo grado di conoscenza. Ogni elemento di questo training set possiede una specifica (label) che indica la corretta azione che il sistema dovrebbe intraprendere in quella situazione, che di solito consiste nell'identificazione di una categoria in cui quella situazione appartiene. L'algoritmo in questo caso cerca di estrapolare e generalizzare le informazioni e la conoscenza contenuta nel training set per poi essere utilizzata per identificare le label corrette non presenti negli insiemi di test. Questo è un importante metodo di learning ma da solo non è adeguato per imparare dalle interazioni. Nei problemi interattivi è molto spesso impraticabile ottenere esempi di comportamenti desiderati che siano sia corretti che rappresentativi di tutte le situazioni nel quale l'agente potrebbe operare. Nelle situazioni e in territori inesplorati un agente dovrebbe poter imparare dalla sola propria esperienza. Il reinforcement learning si differenzia anche dal **learning non supervisionato** che consiste tipicamente nell'estrapolare conoscenza e asso-

ciazioni da dati non classificati.
Learning supervisionato e non supervisionato non esauriscono le tecniche e le metodologie adottabili in machine learning. Nel reinforcement learning infatti si cerca di massimizzare un segnale di reward invece di provare ad estrapolare conoscenza nascosta dai dati. Consideriamo quindi il RL come un terzo paradigma del machine learning.
Una delle challenge di questo paradigma è dato dal trade-off tra esplorazione e sfruttamento. Per ottenere molti reward, o reward alti, l'agente deve preferire azioni che ha già provato in passato e per cui ha già ottenuti reward elevati. Ma per scoprire queste azioni, deve provare azioni che non ha mai selezionato prima. Quindi l'agente deve sia sfruttare quello che ha già avuto modo di apprendere per ottenere il reward, ma anche esplorare per selezionare migliori azioni nel futuro.

1.1.1 Elementi di Reinforcement learning

Oltre agli agenti e all'ambiente, possiamo identificare quattro sotto elementi nel paradigma di reinforcement learning: policy, segnale di reward, funzione valore e, opzionale, un modello dell'ambiente. La **policy** definisce il modo con cui l'agente apprende e il suo comportamento in un dato istante. Per semplificare, una policy può essere vista come il mapping tra gli stati percepibili dall'ambiente e le azioni da prendere quando si è in tali stati. In alcuni casi la policy può essere espressa come semplice funzione o tabella di lookup, mentre in altri può coinvolgere computazione estensiva come processi di ricerca. La policy è il cuore di un agente di RL nel senso che da sola determina il suo comportamento. In generale, le policy possono essere deterministiche, ovvero dipendere solamente dallo stato, o stocastiche, ovvero definite attraverso distribuzione di probabilità sulle azioni, dato uno stato.
Un segnale di **reward** definisce il goal in un problema di reinforcement learning. In ogni time step, l'ambiente invia all'agente un singolo valore chiamato appunto reward. L'unico obiettivo dell'agente è quello di massimizzare il reward totale che riceve durante un episodio del programma. Il segnale di reward definisce così quali sono i buoni ed i cattivi comportamenti per l'agente. Questo segnale è la base di partenza per alterare in modo appropriato la policy; se un azione selezionata dalla policy è seguita da un reward basso, allora la policy potrebbe cambiare per selezionare qualche altra azione in una situazione futura. In generale quindi, i segnali di reward possono essere funzioni stocastiche dello stato dell'ambiente e dell'azione presa.

Mentre il reward indica cosa è buono nell'immediato, la **funzione valore** specifica cosa è buono nel lungo termine. Il valore dello stato è l'ammontare totale dei reward che un agente si aspetta di accumulare nel futuro, partendo da quello stato. Il valore quindi determina la desiderabilità a lungo termine degli stati dopo aver preso in considerazione gli stati migliori da seguire e i reward disponibili in questi stati. Per esempio uno stato potrebbe sempre portare un reward immediato basso ma avere ancora un valore alto poichè è regolarmente seguito da altri stati che portano reward alti, e viceversa. Per fare un analogia umana, i reward sono come il piacere (reward alto) o il dolore (reward basso).
I reward sono in un certo senso, elementi primari, mentre i valori, come predizione dei reward, sono secondari. Senza reward non ci possono essere valori e l'unico scopo di stimare i valori è quello di raggiungere reward più alti. Tuttavia, sono i valori gli elementi di confronto utilizzati quando prendiamo e valutiamo le decisioni. Le scelte delle azioni sono fatte sulla base di giudizi sui valori. Cerchiamo azioni che producano stati a massimo valore, non a massima ricompensa, perché queste azioni ottengono maggior ricompensa nel lungo periodo. Sfortunatamente, è molto più difficile determinare i valori che determinare i reward. I reward sono fondamentalmente dati direttamente dall'ambiente mentre i valori devono essere stimati e ri-stimati dalle sequenze di osservazioni che un agente compie durante la sua intera esistenza. In effetti, la componente più importante di quasi tutti gli algoritmi considerati è un metodo per stimare in modo efficiente i valori, anche se tale ragionamento non è detto sia valido per qualsiasi ambiente. Il quarto ed ultimo elemento è il **modello** dell'ambiente. Per modello si intende un'entità in grado di simulare il comportamento dell'ambiente. Per esempio, data una coppia stato-azione, il modello può predire il risultato della prossima coppia stato-azione. I modelli sono usati per pianificare, ovvero decidere prima quali azioni saranno eseguite sulla base degli stati che potranno essere raggiunti.

1.1.2 Markov decision process

Se si vuole utilizzare un metodo di apprendimento automatico bisogna dare una descrizione formale dell'ambiente. Non interessa sapere esattamente com'è fatto l'ambiente, interessa piuttosto fare delle ipotesi generali sulle proprietà che l'ambiente possiede. Nel reinforcement learning si assume di solito che l'ambiente possa essere descritto da un Processo di Decisione Markoviano (Markov Decision Process o MDP). Un Markov Decision Process è formalmente definito da:

- un insieme finito di stati S.
- un insieme finito di azioni A.
- una funzione di transizione T ($T : S \times A \to \Pi(S)$) che assegna ad ogni coppia stato-azione una distribuzione di probabilità su S.
- una funzione di rinforzo (o reward) R ($R : S \times A \times S \to \Re$) che assegna un valore numerico ad ogni possibile transizione.

Come in parte già accennato, l'interazione stato-ambiente avviene nel seguente modo: all'istante t l'agente percepisce l'ambiente come $s_t \in S$. Sulla base di s_t decide di agire con $a_t \in A$ e l'ambiente risponde dando all'agente una ricompensa immediata (reward) $r_{t+1} = r(s_t, a_t)$ e producendo lo stato successivo $s_{t+1} = \delta(s_t, a_t)$. Le funzioni δ e r sono parte dell'ambiente e non sono necessariamente note all'agente (e possono essere non-deterministiche).
Rispetto al reinforcement learning si dice che un ambiente (un problema) soddisfa l'ipotesi Markoviana (Markov Property) quando:

$$P[s_{t+1} = s, r_{t+1} = r|s_t, a_t] = P[s_{t+1} = s, r_{t+1} = r|s_t, a_t, r_t, \ldots r_1, s_0, a_0] \qquad (1.1)$$

L'ipotesi Markoviana implica che quello che succede all'istante $t + 1, (s_{t+1}, r_{t+1})$ dipende solo da quello che è successo all'istante precedente (s_t, r_t) e non da tutto ciò che è successo in precedenza $(st, at, rt, \ldots r1, s0, a0)$. Se un problema soddisfa l'ipotesi Markoviana ha una dinamica che può essere descritta con una "dinamica ad un passo": ciò significa che è possibile trattare il problema come episodico, dove gli episodi sono i ripetuti tentativi di raggiungere un determinato obiettivo stabilito dal problema. La proprietà di Markov è molto importante nel reinforcement learning perchè tutti i metodi legati a questo tipo di apprendimento basano le proprie scelte assumendo che i valori forniti dall'ambiente siano solamente in funzione dello stato corrente e dell'azione intrapresa all'istante precedente.

1.1.3 Valore di ritorno

L'obiettivo di un agente di RL è quello di scegliere una policy che massimizzi la somma dei reward attesa. La somma dei reward viene chiamata valore di ritorno (G_t) ed è data da:

$$G_t = r_t + r_{t+1} + r_{t+2} + ... + r_{N-1} \tag{1.2}$$

Per compiti a valore continuo, viene definito il *discount return* che è dato da:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1.3}$$

dove $\gamma \in [0,1)$ è chiamato fattore di discount.

1.1.4 Funzione Valore

Per decidere quale azione intraprendere in un certo istante, è importante per l'agente conoscere quanto è "buono" essere in un particolare stato. Un modo per misurare questa bontà dello stato è la funzione valore. Viene definita come la somma dei rewards attesa (E_π) che l'agente riceverà mentre segue una particolare policy π partendo da un particolare stato s. La funzione valore, $V\pi(s)$ per la policy π è data da:

$$V\pi(s) = E_\pi(G_t|s_t = s) = E_\pi(\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s) \tag{1.4}$$

Similmente, una **funzione azione-valore** (action value function), chiamata anche **Q-function**, può essere definita come la somma dei rewards prevista mentre si intraprende un'azione a in uno stato s, seguendo la policy π. La funzione azione-valore $Q_\pi(s,a)$ è definita come segue:

$$Q\pi(s,a) = E_\pi(G_t|s_t = s, a_t = a) = E_\pi(\sum_{k=0}^{\infty} \gamma^k r_{t+k}|s_t = s, a_t = a) \tag{1.5}$$

1.1.5 Equazioni di Bellman

Le equazioni di Bellman formulano il problema di massimizzazione della somma dei reward attesa in termini di relazione ricorsiva con la funzione valore. Una policy π è considerata migliore di un altra policy π' se il ritorno atteso di quella policy è maggiore di π' per tutti gli $s \in S$, che implica $V^\pi(s) \geq V^{\pi'}(s)$ per tutti gli $s \in S$. Quindi la funzione valore ottimale $V_*(s)$ può essere definita come

$$V_*(s) = \max_\pi V_\pi(s), \ \forall s \in S \tag{1.6}$$

Similmente, la funzione azione-valore ottimale $Q_*(s,a)$ può essere definita come

$$Q_*(s,a) = \max_\pi Q_\pi(s,a),\ \forall s \in S,\ a \in A \tag{1.7}$$

Inoltre, per una policy ottimale, possiamo definire l'equazione

$$V_*(s) = \max_{a \in A(s)} Q_{\pi^*}(s,a) \tag{1.8}$$

Espandendo l'equazione 1.8 con 1.5 otteniamo

$$\begin{aligned} V_*(s) &= \max_a E_{\pi^*}(G_t | s_t = s, a_t = a) \\ &= \max_a E_{\pi^*}(r_t + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a) \\ &= \max_a \sum_{s'} p(s'|s,a)[r_t + \gamma V_*(s')] \end{aligned} \tag{1.9}$$

Il primo passaggio indica la scomposizione ricorsiva di G_t (equazione 1.3), mentre il secondo passaggio pone

$$E_\pi[\ ] = \sum_{s'} p(s'|s,a)[\ ] \tag{1.10}$$

dove $p(s'|s,a)$ indica la probabilità di arrivare nello stato s' partendo dallo stato s ed intraprendendo l'azione a.
L'equazione 1.9 è conosciuta come equazione di ottimalità di Bellman per $V_*(s)$. Mentre per Q_* l'equazione di ottimalità è

$$\begin{aligned} Q_*(s,a) &= E(r_t + \gamma \max_{a'} Q_*(s_{t+1}, a') | s_t = s, a_t = a) \\ &= \sum_{s'} p(s'|s,a)[r_t + \gamma \max_{a'} Q_*(s', a')] \end{aligned} \tag{1.11}$$

dove $Q_*(s,a)$ è definita ricorsivamente in base alle equazioni 1.5 e 1.7 a cui viene applicata la trasformazione in 5.1.
Se sono conosciute le probabilità di transizione e le funzioni di reward, le equazioni di ottimalità di Bellman possono essere risolte in modo iterativo. Questo approccio è conosciuto come **programmazione dinamica**. Gli algoritmi che presuppongono che queste probabilità siano conosciute, o che vengano stimate sul momento, sono conosciuti come algoritmi *model-based*. Ma per molti altri algoritmi si assume che le probabilità non siano conosciute e vanno stimate sia policy che funzione valore attraverso *rollout* del sistema, ovvero applicando uno o più step di simulazione per testare le possibili conseguenze. Questi metodi sono conosciuti come algoritmi

model-free. Monte Carlo, Temporal Difference e Policy Search sono gli algoritmi model-free più comunemente usati. Il resto di questo capitolo si focalizzerà su quest'ultima classe di metodi per introdurre gli algoritmi effettivamente utilizzati: Q-learning e, nel successivo capitolo, Deep Q-Learning.

1.1.6 Metodi model-free

I metodi model-free possono essere applicati a molti problemi di reinforcement learning che non richiedono alcun modello dell'ambiente. Molti approcci model-free cercano di apprendere la funzione valore e da essa inferire la policy ottimale oppure ricercando la policy ottimale direttamente nello spazio dei parametri della policy stessa. Questi approcci possono essere classificati anche come approcci on-policy o approcci off-policy. I metodi on-policy utilizzano la policy corrente per generare azioni e la utilizzano per aggiornare la policy stessa mentre i metodi off-policy utilizzano una policy di esplorazione diversa per generare azioni rispetto alla policy che viene aggiornata.

Metodi Monte Carlo

I metodi Monte Carlo lavorano sull'idea della GPI (generalized policy iteration). La GPI è uno schema iterativo ed è composto da due processi. Il primo prova a costruire un'approssimazione della funzione valore basandosi sulla policy corrente (policy evaluation step). Nel secondo step, la policy viene migliorata rispetto alla funzione a valore corrente (policy improvement step). Nei metodi Monte Carlo, per stimare la funzione valore si utilizza la tecnica del rollout eseguendo la policy corrente sul sistema. La funzione valore viene poi stimata utilizzando il reward accumulato sull'intero episodio e la distribuzione degli stati incontrati. La policy corrente è quindi stimata attraverso tecnica greedy. Utilizzando questi due step in modo iterativo, è possibile dimostrare che l'algoritmo converge alla funzione e alla policy del valore ottimale. Sebbene i metodi Monte Carlo siano semplici nella loro implementazione, richiedono un gran numero di iterazioni per la loro convergenza e soffrono di una grande varianza nella stima della funzione valore.

Metodi Temporal Difference

I metodi Temporal Difference (TD) sono costruiti sull'idea della GPI ma differiscono dai metodi Monte Carlo nel evaluation step della policy. Invece di usare la somma

totale di reward, questi metodi calcolano l'errore temporale, ovvero la differenza tra la nuova stima e la vecchia stima della funzione valore, considerando il reward ricevuto al time step corrente e utilizzandolo per aggiornare la funzione valore. Questo tipo di aggiornamento riduce la varianza ma incrementa il bias nella stima della funzione valore. L'equazione di aggiornamento della funzione valore è data da:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') + V(s)] \tag{1.12}$$

dove α è il fattore di learning (learning rate), r è il reward ricevuto nell'istante corrente, s' è il nuovo stato e s è il vecchio stato. Quindi i metodi temporal difference aggiornano la funzione valore ad ogni time step, al contrario dei metodi Monte Carlo che aspettano il completamento dell'episodio per aggiornare la funzione valore.

SARSA

I due algoritmi di TD più diffusi per risolvere problemi di reinforcement learning sono SARSA e Q-Learning. SARSA è un metodo temporal difference on-policy, ovvero tenta di apprendere la funzione azione-valore invece della funzione valore. Lo step di evaluation usa l'errore temporale per la funzione azione-valore, come avviene similmente per la funzione valore. L'algoritmo è descritto come segue:

Algorithm 1 SARSA

Inizializza Q(s,a) random
repeat
  Osserva stato iniziale s_1
  Seleziona un azione a_1 usando la policy derivata da Q (es: ϵ-greedy)
  for t=1 **to** T **do**
    Esegui azione a_1
    Osserva il reward r_t e il nuovo stato s_{t+1}
    Scegli la nuova azione a_{t+1} usando la policy derivata da Q (es: ϵ-greedy)
    Aggiorna Q usando
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
  end for
until terminazione

Q-Learning

Watkins [2] instruce un algoritmo temporal difference off-policy conosciuto come Q-Learning. Q-Learning, a differenza di SARSA, è off-policy poichè approssima direttamente Q_* indipendentemente dalla policy che sta seguendo. Un esperienza è definita come (s, a, r, s') dove l'agente parte nello stato s, esegue l'azione a, riceve un reward r, e si muove in un nuovo stato s'. L'update su $Q(s, a)$ è dato quindi ricevendo il reward massimo possibile da un'azione da s' e applicando l'aggiornamento:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t)] \tag{1.13}$$

L'algoritmo è descritto come segue:

Algorithm 2 Q-Learning

Inizializza Q(s,a) random
repeat
  Osserva stato iniziale s_t
  for t=1 **to** T **do**
    Scegli un azione a_t usando la policy derivata da Q (es: ϵ-greedy)
    Esegui azione a_t
    Osserva il reward r_t e il nuovo stato s_{t+1}
    Aggiorna Q usando 1.13
  end for
until terminazione

Il metodo più semplice per salvare i valori della funzione valore per ognuno dei differenti stati è la forma tabellare, anche se questa forma presenta alcune limitazioni: se lo spazio degli stati del problema è molto vasto risulta impossibile salvare tutti i valori nel formato tabellare. La ragione è molto semplice: la memoria richiesta per salvare tutti questi dati è troppo vasta. In aggiunta, anche la sola ricerca nella tabella di un valore in un particolare stato potrebbe essere computazionalmente proibitiva. Un'altra limitazione è dovuta allo spazio degli stati: se lo spazio è continuo risulterà impossibile utilizzare la forma tabellare, a meno di una discretizzazione degli stati stessi. Per questo motivi l'adozione del formato tabellare viene applicata solamente ad ambienti con ridotto numero di stati e azioni.
Per superare questi problemi sono stati introdotti approssimatori di funzione per salvare la funzione valore. La funzione valore è parametrizzata da un vettore $\theta = (\theta_1, \theta_2, ..., \theta_n)^T$ ed è indicato con $V(s; \theta^V)$. L'approssimatore di funzione può essere pensato come ad un mapping tra il vettore θ in $\mathbb{R}^n$ con lo spazio della funzione valore. Finché il numero dei parametri dell'approssimatore è minore del numero di valori di stato, il cambiamento di un valore di un certo parametro comporta il cambiamento della funzione valore in più regioni dello spazio degli stati. Questo aiuta gli approssimatori di funzione ad effettuare una miglior generalizzazione in un numero di passi di training minore.
Esistono vari metodi in Reinforcement learning per l'approssimazione di funzione. In questa tesi per semplicità verranno trattare solamente le Reti Neurali in maniera generale, per poi focalizzare l'attenzione sull'algoritmo di Deep Q-Learning.

1.2 Reti neurali

Le reti neurali artificiali (Artificial Neural Network) sono uno degli strumenti attualmente più utilizzati per risolvere problemi complessi. Questi modelli, ispirati alla natura, possono essere definiti da una collezione di unità di processo densamente interconnesse tra loro chiamate neuroni, che lavorano all'unisono per portare a termine il lavoro di computazione. Quello che rende le ANN particolarmente interessanti è la possibilità di riprodurre molte delle caratteristiche desiderabili del cervello umano. Queste includono la capacità al parallelismo, abilità di apprendimento, di generalizzazione, adattatività, tolleranza a failure e basso consumo di energie [3]. Un neurone è una speciale cellula biologica che processa informazioni. È composta da un corpo (*Soma*) e da due tipologie di ramificazioni: i *dendriti* che trasportano il segnale nervoso ricevuto da altri neuroni verso il soma, e l'*assone* che conduce il segnale nervoso. I dendriti fungono quindi da ricevitori del segnale nervoso proveniente dagli altri neuroni, mentre l'assone funge da trasmettitore verso i neuroni successivi. I dendriti ricevono impulsi elettrochimici da altri neuroni che passano attraverso il soma e quindi attraverso l'assone. Al termine dell'assone è presente una struttura detta giunzione sinaptica che permette il trasferimento di questi impulsi alla cellula neurale successiva. Dato il vasto numero di dendriti e connessioni sinaptiche, il neurone biologico è in grado di ricevere molti segnali simultaneamente.

Neuroni artificiali

Il cervello umano è quindi formato da una complessa rete di neuroni interconnessi che hanno l'abilità di estrarre informazione critica da un segnale di input e produrre un determinato segnale di output. Similmente, una rete neurale artificiale consiste anch'essa di una rete di neuroni interconnessi, suddivisi in livelli dove ogni neurone prende segnali in input e restituisce un segnale di output.

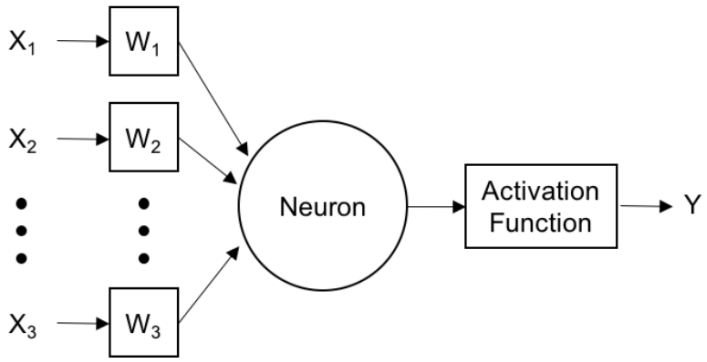


Figura 1.2: Modello di neurone artificiale

La figura 1.2 mostra un neurone artificiale. Il neurone prende un vettore di input x e calcola la somma pesata degli input, dove il peso è indicato con w. La somma pesata viene aggiunta al bias b e passata ad una funzione di attivazione f, che produce l'output del neurone. L'equazione che rappresenta il funzionamento del neurone può essere definita come segue:

$$y_i = f(\sum_j x_j w_{i,j} + b_1) \tag{1.14}$$

Funzione di attivazione

La funzione di attivazione introduce non linearità nell'output del neurone. Questo aiuta la rete ad apprendere rappresentazioni non lineari dai dati di input. In letteratura sono presenti varie funzioni di attivazione, come la *sigmoide*, la *tangente iperbolica* e la *rectified linear unit* (ReLu).
Sigmoide:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1.15}$$

Tangente iperbolica:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.16}$$

ReLu:

$$f(x) = \max(0, x) \tag{1.17}$$

Feed-Forward Networks

L'architettura più comunemente usata per creare la rete neurale è la Feed-Forward Network. La figura 1.3 mostra un esempio di questa architettura. I neuroni sono

organizzati tipicamente in livelli (layer), e in particolare di tre tipi: l'input layer, uno o più livelli nascosti (hidden layer) e un output layer. Il flusso di informazione parte dall'input layer, si propaga ai livelli nascosti, ed infine arriva all'output layer che computa l'output finale. Ogni neurone nei differenti livelli applica la stessa computazione (equazione 1.14). Nelle reti feed-forward l'output di ogni neurone di un determinato livello è connesso a tutti gli input dei neuroni al livello successivo e non ci sono cicli.
Nelle reti feed-forward l'output di ogni neurone di un determinato livello è connesso a tutti gli input dei neuroni al livello successivo e non ci sono cicli.
I pesi dei neuroni della rete sono modificati in base ad una tecnica conosciuta come *backpropagation*. L'idea di questo approccio è di partire con un inizializzazione random dei pesi e calcolare l'output per un dato input. L'errore tra l'output generato e l'output attuale viene utilizzato per aggiornare i pesi della rete con un algoritmo di gradient descent. Quando le Artificial neural network presentano più di un hidden layer prendendono il nome di **Deep neural network** (DNN).

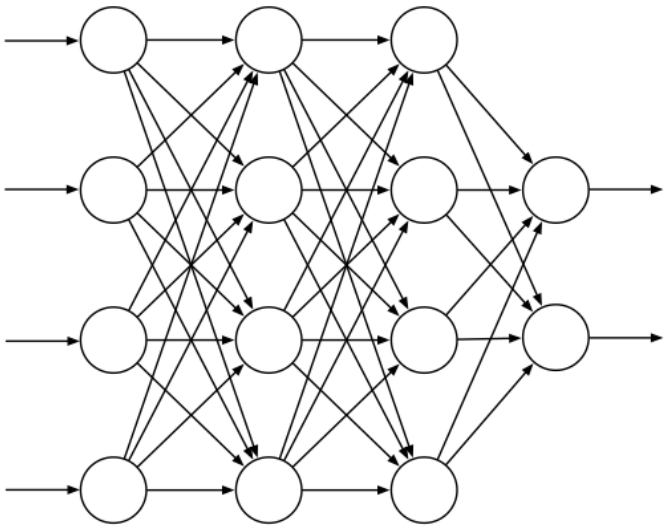


Figura 1.3: Modello di rete neurale con 4 input, 2 hidden layer con 4 neuroni ciascuno, e 2 neuroni in output

1.3 Deep Reinforcement Learning

Gli algoritmi di Deep reinforcement learning (RDL) hanno ricevuto, e stanno ricevendo, un notevole interesse da parte della comunità AI (Artificial intelligence) negli ultimi anni. Con il termine Deep Reinforcement learning ci si riferisce semplicemente all'uso di Deep Neural networks come approssimatori di funzione per la funzione valore o la policy, in algoritmi di Reinforcement learning. È stato dimostrato come algoritmi di Deep reinforcement learning hanno raggiunto, e in alcuni casi superato, prestazioni a livello umano nel giocare a videogiochi Atari [4].
In questo paragrafo viene spiegato nel dettaglio l'algoritmo di Deep Q-learning utilizzato poi negli ambienti virtuali realizzati.

1.3.1 Deep Q-Learning

Q-Learning è un algoritmo largamente usato nel reinforcement learning. Inizialmente era considerato un algoritmo instabile quando usato con le reti neurali e quindi il suo utilizzo veniva limitato a compiti e problemi che coinvolgevano spazi di stati a limitata dimensionalità. E stato però dimostrato in [4] che algoritmi e tecniche di Q-Learning possono essere utilizzate con le DNN. Questo algoritmo ha mostrato il raggiungimento di performance di livello umano su sette videogames su console Atari 2600 utilizzando solamente immagini di pixel grezze come input. In questo paper ci si riferisce a tale algoritmo con il nome di *Deep Q-Learning* o *Deep Q-Network* (DQN).
DQN è un metodo di RL di approssimazione di funzione. Rappresenta un'evoluzione del metodo Q-Learning dove la tabella stato-azione viene sostituita da una rete neurale. In questo algoritmo quindi l'apprendimento non consiste nell'aggiornare la tabella ma consiste nell'aggiustamento dei pesi dei neuroni che compongono la rete attraverso backpropagation. L'apprendimento della funzione valore in DQN è basato quindi sulla modifica dei pesi in funzione della loss function:

$$L_t = (E[r + \gamma \max_a Q(s_{t+1}, a_t)] - Q(s_t, a_t))^2 \tag{1.18}$$

dove $E[r+\gamma \max_a Q(s_{t+1}, a_t)]$ rappresenta l'expected return ottimo mentre $Q(s_t, a_t)$ è il valore stimato dalla rete.
Gli errori calcolati dalla loss function saranno propagati al'indietro nella rete mediante un passo backword (backpropagation), seguendo la logica di discesa del gradiente. Infatti il gradiente indica la direzione di maggior crescita di una funzione

e muovendoci in direzione opposta riduciamo (al massimo) l'errore. Il comportamento della policy è dato da un approccio e-greedy per garantire un esplorazione sufficiente. L'aspetto chiave che fa funzionare le DQN è l'utilizzo dell'*experience replay*. Con questa tecnica, l'esperienza dell agente $e_t = (s_t, a_t, r_t, s_{t+1})$ viene presa ad ogni time step t e salvata in un data-set $D = e_t, e_{t+1}, \ldots, e_n$ chiamato *replay memory*. Il training viene effettuato attraverso tecnica a mini-batch, ovvero prelevando un sotto-insieme di campioni di esperienze estratte random da questo replay memory. L'utilizzo di questa tecnica permette l'utilizzo delle esperienze passate di essere usate in più di un aggiornamento della rete. In più il sotto-insieme scelto in maniera casuale dalla replay memory permette di interrompere la forte correlazione presente tra esperienze successive riducendo così la varianza tra gli aggiornamenti.

Algorithm 3 Deep Q learning con Experience Replay

Inizializza Replay Memory D
Inizializza Q(s,a) con pesi random
repeat
  Osserva stato iniziale s_1
  for t=1 **to** T **do**
    Seleziona un azione a_t usando Q (es: ϵ-greedy)
    Esegui azione a_t
    Osserva il reward r_t e il nuovo stato s_{t+1}
    Salva la transizione (s_t, a_t, r_t, s_{t+1}) nel Replay Memory D
    Preleva un campione di transizioni (s_j, a_j, r_j, s_{j+1}) da D
    Calcola il target T per ogni transizione
    if s_{j+1} è Terminale **then**
      $T = r_j$
    else
      $T = r_j + \gamma \max_a Q(s_{j+1}, a_j)$
    end if
    Addestra la rete Q minimizzando $(T - Q(s_j, a_j))^2$
  end for
until terminazione

Capitolo 2

Ambiente di simulazione

La computer grafica 3D è un ramo della computer grafica che si basa sull'elaborazione di un insieme di modelli tridimensionali tramite algoritmi atti a produrre una verosimiglianza fotografica e ottica nell'immagine finale. Essa viene utilizzata nella creazione e post produzione di opere per il cinema, televisione, nei videogiochi, nell'architettura, nell'ingegneria, nell'arte e in svariati ambiti scientifici. In particolare il capitolo affronta il problema della simulazione di un ambiente reale. Molte volte è impossibile o impraticabile sperimentare direttamente sul campo (studi in assenza di gravità) oppure la sperimentazione richiederebbe troppo tempo e risorse (training di algoritmi) quindi la simulazione ci viene in aiuto per accellerare e rendere possibili questi studi.
Questo capitolo affronta il tema della simulazione e introduce il framework ***Bullet Phisics*** utilizzato per creare e simulare gli ambienti presentati in questa tesi.

2.1 Bullet Physics

Bullet è un physics engine ovvero un modulo software che simula un modello fisico newtoniano utilizzando variabili come massa, velocità, attrito alla resistenza del vento e altro. Il motore, utilizzando questi dati e le leggi newtoniane, simula il comportamento degli oggetti sottoposti alle forze del mondo (reale o immaginario). Bullet è open source, realizzato in linguaggio C++ e la sua versione in linguaggio Python su cui è basato il lavoro qui proposto è ***pyBullet***. PyBullet viene impiegato per la simulazione fisica di robot, giochi, effetti visivi e machine learning. PyBullet offre la possibilità di utilizzare oggetti 3D in diversi formati e fornisce meccanismi sofisticati come la simulazione dinamica forward, cinematica inversa, collision

detection, inoltre fornisce l'integrazione con Gym openAI, toolkit in Python per gestire in maniera semplice ambienti simulati utilizzabili per implementare tecniche di machine learning.
Oltre alla simulazione, il framework fornisce supporto alla realtà virtuale e integrazione con HTC Vive e Oculus Rift.

Figura 2.1: Esempio di render grafico dell'ambiente Bullet

2.1.1 OpenGL

Bullet sfrutta le API fornite da OpenGL [6] per il rendering del proprio ambiente. OpenGL (*Open Graphics Library*) è un interfaccia software per il motore grafico. È una libreria di grafica e modellazione 3D, molto veloce e performante. Sono fornite implementazioni di OpenGL per le principali piattaforme e sistemi operativi inclusi ovviamente Windows, Mac OS e Linux. OpenGL è progettato per essere utilizzato con hardware dedicato alla visualizzazione e manipolazione di grafica 3D. OpenGL utilizza un approccio procedurale anzichè descrittivo. Ciò significa che invece di descrivere la scena e come dovrebbe apparire, il programmatore scrive passo dopo passo le operazioni necessarie per creare l'aspetto desiderato.

2.2 Terminologia

In questa sezione vengono descritti alcuni termini utilizzati quando si parla di motori fisici.

2.2.1 Physics engine

Un physics engine (motore fisico) è la componente di un programma che si occupa di computare come gli oggetti si devono comportare. Per simulare questi comportamenti entrano in gioco attributi come massa, velocità e attrito. I motori fisici possono simulare una varietà di entità fisiche come corpi rigidi, tessuti e liquidi. La simulazione fisica può essere suddivisa in due fasi, detection delle collisioni e simulazione dinamica.

2.2.2 Rappresentazione degli oggetti

Gli oggetti in un physics engine possono essere rappresentati in modi diversi. È uso comune rapresentare gli oggetti attraverso primitive, come sfere, cilindri o cuboidi. Possono essere impiegate anche mesh triangolari per rappresentare qualsiasi tipo di forma.

2.2.3 Attrito

L'attrito è una forza che si verifica quando due oggetti sono in contatto e si muovono. La forza di attrito contrasta il loro movimento e lavora in una direzione perpendicolare alla normale forza delle superfici di contatto.

2.2.4 Vincoli

Un corpo rigido ha sei gradi di libertà quando non è legato a vincoli: tre gradi di transazione e tre di rotazione. La libertà di transazione permette all'oggetto di muoversi in ognuna delle tre dimensioni, mentre la libertà di rotazione consente all'oggetto di cambiare l'angolo su cui ruotare attorno ai tre assi. Un vincolo è un modello che rimuove i gradi di libertà. Ad esempio un giunto a perno è un vincolo che forza un oggetto a ruotare attorno a un asse, quindi ha uno solo grado di libertà. Nella sezione successiva sono illustrati nel dettaglio le tipologie di giunti/vincoli.

2.2.5 Collision detection

La collision detection, ovvero la gestione delle collisioni tra gli oggetti, viene eseguita da un collision dispacher iterando ogni coppia di oggetti ed applicando algoritmi di collisione in base ai diversi tipi di **oggetti collisione** coinvolti per calcolare i punti di contatto. Gli oggetti collisione sono oggetti costituiti da una o più figure di collisione. Figure base come cuboidi, sfere, cilindri, coni possono essere combinate per ottenere forme più avanzate. Il filtraggio delle collisioni è un meccanismo molto utile in Bullet per garantire che solo alcuni oggetti possano scontrarsi tra loro. Sono supportate maschere bit a bit come modalità per decidere se gli oggetti di collisione possano scontrarsi con altri oggetti di collisione.

2.2.6 Dinamica dei corpi rigidi

Il corpo rigido viene derivato dall'oggetto di collisione (o viceversa) con l'aggiunta di proprietà quali forze, massa, inerzia, velocità e vincoli. Vi sono tre tipi differenti di corpi rigidi. I corpi rigidi *dinamici* sono corpi rigidi che possiedono massa positiva e aggiornano il proprio stato durante ogni step di simulazione. I corpi rigidi *statici* invece possiedono massa pari a zero e non possono muoversi ma possono subire collisioni con altri corpi. i corpi rigidi *statici con cinematica* sono simili a questi ultimi, ovvero hanno massa uguale a zero, ma possono essere animati dall'utente.

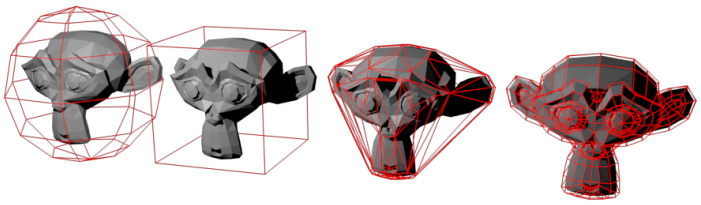


Figura 2.2: Esempio di oggetti collisione (forma geometrica rossa) e corpo rigido

2.2.7 Metodi di simulazione

Il cuore di un motore fisico è chiamato solver o stepper. Il solver, dato un time step, produce un passo nella simulazione, ovvero computa nuove posizioni, velocità lineari, velocità angolari dei corpi presenti nella scena e deve tenere in considerazione tutte le collisioni e i vincoli imposti.

2.3 PyBullet in machine learning

Lo scopo di studio di questa libreria è quello di creare un ambiente virtuale su cui verranno testati ed impiegati algorimi di machine learning, in particolare di reinforcement learning, ovvero sistemi in grado di apprendere ed adattarsi alle mutazioni dell'ambiente in cui sono immersi attraverso la distribuzione di una "ricompensa" detta *reward* che consiste nella valutazione delle loro prestazioni. (capitolo 1)

2.3.1 Gym OpenAI

PyBullet [7] può essere facilmente usato con TensorFlow [8] e fornisce un integrazione nativa con il framework Gym OpenAI [9].
Gym OpenAI è una collezione di environment creati per il testing e lo sviluppo di algoritmi di reinforcement learning molto utili allo sviluppatore che non deve preoccuparsi di realizzare da zero ambienti complessi. Sul sito è presente anche una leaderboard in cui è possibile sottomettere il codice realizzato per risolvere un particolare problema e visualizzare i risultati degli altri utenti.
L'integrazione con pyBullet è forte tanto è che viene utilizzata la stessa struttura di base di Gym per realizzare gli ambienti di simulazione.

2.3.2 Struttura di base

OpenAI Gym si basa sui fondamentali del reinforcement learning e fornisce una struttura che rispecchia l'interazione tra agenti e ambiente: agente interagisce in un environment intraprendendo azioni e provando a massimizzare un certo reward che viene rilasciato a fronte delle azioni intraprese.
La struttura è pertanto di natura **episodica**. La classe principale `Env` può essere utilizzata seguendo questa struttura di base:

```
env = gym.make('CartPole-v1')
   maxSteps = 1000
   for x in maxSteps:
      done = False
      state = env.reset()
      while not done:
         action_result = action(state) # calcola l'azione
```

```
        state, reward, done, info = env.step(action_result)
        env.render()
```

il metodo `make` inizializza l'ambiente, `reset` lo resetta e restituisce lo stato iniziale, `step` esegue l'azione che gli viene passata come parametro e restituisce il nuovo stato corrente mentre `render` renderizza e mostra l'agente nel nuovo stato. Il metodo `step` restituisce anche il `reward` e un flag `done` che indica il completamento dell'obiettivo da parte dell'agente o l'impossibilità di proseguire (game over).

2.4 Il motore grafico

Utilizzando la classe `Env` proposta dal framework come base per i nuovi ambienti, si adotta l'interfaccia comune prevista dal toolkit. In questo modo algoritmi già realizzati per risolvere altri problemi potranno essere utilizzati anche per ambienti appena realizzati con modifiche minime o nulle. Ai metodi dell'interfaccia comune sono poi aggiunte le chiamate necessarie a definire l'interfaccia grafica fornita dal modulo python pybullet.
Dopo aver importato il modulo, la prima cosa da fare è connettersi al motore fisico. pybullet è progettato attorno ad un'API client-server-driven, con un client che invia comandi ed un server fisico che restituisce lo stato. Pybullet ha alcuni server fisici integrati: DIRECT e GUI. Entrambe le connessioni GUI e DIRECT eseguiranno la simulazione fisica e il rendering nello stesso processo di pybullet.
La connessione DIRECT invia i comandi direttamente al motore fisico, senza utilizzare alcun livello di trasporto e nessuna finestra di visualizzazione grafica, e restituisce direttamente lo stato dopo l'esecuzione del comando.
La connessione GUI crea invece una nuova interfaccia grafica utente (GUI) con rendering 3D OpenGL, all'interno dello stesso spazio di processo di pybullet. Su Linux e Windows questa GUI viene eseguita in un thread separato, mentre su OSX viene eseguita nello stesso thread a causa delle limitazioni del sistema operativo.
I comandi e i messaggi di stato vengono inviati tra il client pybullet e il server di simulazione fisica della GUI utilizzando un normale buffer di memoria. È anche possibile connettersi a un server fisico in un processo diverso sulla stessa macchina o su una macchina remota utilizzando memoria condivisa o attraverso canali TCP/UDP.
Di default non è presente nessuna forza gravitazionale nell'ambiente. *setGravity* è un metodo che ci permette di settare la gravità per tutti gli oggetti presenti nella

scena. Gli oggetti possono essere creati direttamente nella scena oppure vengono importati da file. La creazione è possibile solo per oggetti semplici (cuboidi, sfere ecc) mentre sono molto utili ed utilizzati i metodi forniti per caricare oggetti nella scena. La parte 2 di questo documento introduce i meccanismi di costruzione e configurazione degli oggetti.
La posizione degli oggetti all'interno dell'ambiente è espressa in coordinate nello spazio Cartesiano. L'orientazione (o rotazione) degli oggetti può essere espressa usando quaternioni [x, y, z, w], angoli di eulero o matrici 3×3. PyBullet fornisce metodi per la conversione tra questi formati.

2.4.1 Controllo degli oggetti/robot

I robot vengono descritti come un insieme di solidi (link) connessi attraverso giunti (joint) che fungono da vincoli. Grazie al controllo che può essere esercitato su questi giunti, i robot vengono controllati e possono eseguire azioni all'interno dell'ambiente. Ogni giunto connette un link genitore ad un link figlio in modo gerarchico. Sono sempre presenti quindi $n - 1$ giunti dove n è il numero di link di cui è composto il robot.
`setJointMotorControl2` è il metodo utilizzato per controllare e muovere i joint. Richiede come parametro l'id univoco del robot, l'indice del giunto da controllare, e una modalità di controllo che può essere di posizione, forza di velocità o forza di rotazione.

2.5 Descrizione degli oggetti 3D

2.5.1 Unified Robot Description Format

Gli oggetti 3D, come già accennato, possono essere descritti direttamente all'interno del codice oppure possono essere importati da file. I formati ammissibili sono vari ma quello che viene utilizzato maggiormente è il formato URDF [10]. Il formato **URDF** (*Unified Robot Description Format*) è un file **XML** che descrive un robot (chiameremo robot il nostro oggetto 3D poichè può essere la composizione di più oggetti). Il robot viene descritto attraverso una struttura ad albero e tutte le parti (*link*) che lo compongono devono essere collegati da giunture (*joint*) rigide; gli elementi flessibili non sono supportati.

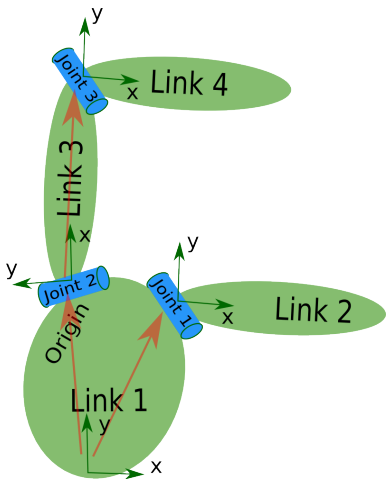


Figura 2.3: robot composto da joint e link

La descrizione del robot consiste quindi in un insieme di elementi *link* e un insieme di elementi *joint* che connettono i *link* tra di loro. Quindi una tipica descrizione di un robot assomiglia a:

```
<robot name="robot1">
        <link> ... </link>
        <link> ... </link>
        <link> ... </link>

        <joint> .... </joint>
        <joint> .... </joint>
```

```
</robot>
```

2.5.2 Componente <link>

la componente **link** descrive un corpo solido tramite le proprietà `inertial` `visual` e `collision`:

```
<link name="my_link">
        <inertial>
          <origin xyz="0 0 0.5" rpy="0 0 0"/>
          <mass value="1"/>
          <inertia ixx="100" ixy="0" ixz="0"
                      iyy="100" iyz="0" izz="100" />
        </inertial>
        <visual>
           <origin xyz="0 0 0" rpy="0 0 0" />
           <geometry>
                <box size="1 1 1" />
           </geometry>
           <material name="Cyan">
                <color rgba="0 1.0 1.0 1.0"/>
           </material>
        </visual>
        <collision>
           <origin xyz="0 0 0" rpy="0 0 0"/>
           <geometry>
                <cylinder radius="1" length="0.5"/>
           </geometry>
        </collision>
</link>
```

`inertial` descrive le proprietà di inerzia del corpo rigido come ad esempio la sua massa e il punto di origine. La proprietà `visual` descrive come il corpo verrà visualizzato (geometria, colori e materiali di cui è composto il corpo) e la proprietà `collision` descrive l'oggetto collisione descritto nel paragrafo precedente.

Gli oggetti possono essere definiti anche attraverso mesh importate da file per poter costruire solidi 3D più complessi.

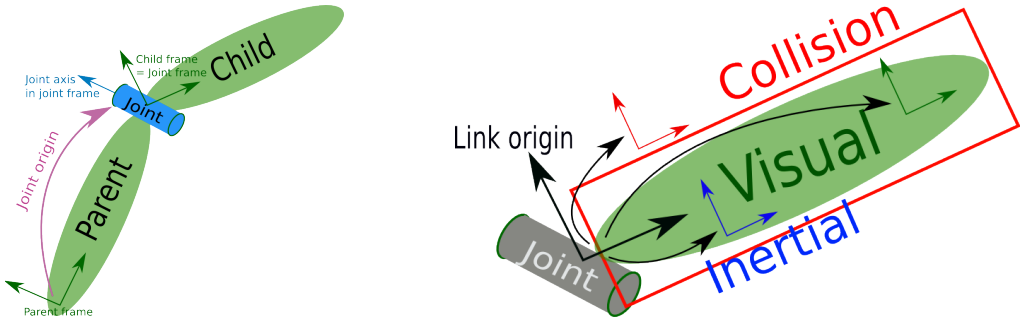
2.5.3 Componente <joint>

La componente **joint** descrive la cinematica e la dinamica della giuntura e ne specifica anche i limiti (limiti di movimento):

```
<joint name="my_joint" type="prismatic">
  <axis xyz="1 0 0"/>
  <origin xyz="0.0 0.0 0.0"/>
  <parent link="link1"/>
  <child link="link2"/>
  <limit effort="1000.0" lower="-15" upper="15" velocity="5"/>
</joint>
```

`parent` e `child` sono i 2 link legati dalla giuntura. L'elemento **joint** ha 2 attributi:

- **name** - specifica il nome univoco del joint.
- **type** - specifica il tipo di joint, che può essere uno dei seguenti:
  - *revolute* -Joint a perno che può ruotare attorno agli assi specificati. Ha un limite di range specificato da upper e lower.
  - *prismatic* - Joint che può scorrere lungo un asse. Ha un limite di range specificato da upper e lower.
  - *fixed* - Questo non è realmente un joint in quanto non può muoversi. Tutti i gradi di liberta solo bloccati.
  - *continuous* - Joint che può ruotare attorno agli assi specificati. Non prevede limiti di range.
  - *floating* - Questo joint permette il movimento per tutti e 6 i gradi di libertà.
  - *planar* - Questo joint permette il movimento nel piano perpendicolare agli assi specificati.



(a) elemento joint (b) elemento link

Figura 2.4: joint e link

2.6 Importanza della simulazione

Perchè la simulazione è importante?

Lo studio di un particolare problema in ambiente simulato rispetto ad un ambiente reale offre notevoli vantaggi. La velocità per il training di modelli in questi environments può essere aumentata considerevolmente, diminuendo quindi quelle che sono le tempistiche reali di osservazione sugli ambienti, cosa che non può essere fatta in un ambiente reale dove la variabile "tempo" non può essere controllata. La simulazione offre la possibilità di ricreare ambienti estremi o difficili da raggiungere come altri pianeti o lo spazio, fornendo quindi supporto al loro studio. Il fattore sicurezza è un altro aspetto cruciale: gli ambienti simulati offrono un ambiente controllato in cui testare gli effetti di particolari azioni. Ad esempio tutto ciò che concerne lo studio sulla guida autonoma è stato prima costruito e testato in simulazione e poi trasposto in ambiente reale per evitare sia costi elevati dovuti ai possibili malfunzionamenti, sia per evitare spiacevoli incidenti.

2.6.1 Simulazione e animazione

Negli ambienti virtuali la credibilità è essenziale. L'utilizzo di vincoli fisici offre una sfida continua anche per l'animazione di personaggi e oggetti che popolano un ambiente virtuale. Tradizionalmente le traiettorie di movimento di personaggi e oggetti virtuali vengono realizzate a mano da animatori esperti. Questo processo richiede tempistiche elevate e le animazioni risultanti sono puramente cinematiche: non sfruttano quindi le proprietà di forza o massa. La validità fisica dei movimenti dipende quindi esclusivamente dall'abilità dell'animatore.
Rendere un'animazione realistica diventa ancora più difficile durante le interazioni.

Vi sono molti modi in cui le entità virtuali possono interagire tra loro e variazioni sottili possono dare risposte sostanzialmente diverse. Ad esempio, una pila di scatole può collassare in infiniti modi, anche con perturbazioni iniziali simili. I sistemi di animazione che derivano da dati esistenti (acquisiti o realizzati a mano) richiedono un processo complesso che coinvolge eventi, regole e elaborazione geometrica per generare risposte adeguate. Nonostante i grandi progressi dei metodi basati sui dati nell'ultimo decennio, la loro capacità di produrre un'animazione reattiva, plausibile e non ripetitiva è limitata. La simulazione basata sulla fisica offre un approccio fondamentalmente diverso all'animazione computerizzata appena descritta. Invece di manipolare direttamente i movimenti di oggetti e personaggi, questo approccio consente a tutti i movimenti di essere il risultato di un processo di simulazione basato sulla fisica. Di conseguenza, i personaggi e gli oggetti interagiscono automaticamente in conformità alle leggi fisiche, senza la necessità di ulteriori dati di movimento o script. Negli ultimi decenni, la simulazione basata sulla fisica è diventata un metodo consolidato per l'animazione dei fenomeni passivi, come vento, movimento di acqua ecc.

Capitolo 3

Ambienti sviluppati

In questo capitolo vengono mostrati nel dettaglio gli ambienti creati sui quali sono stati testati due algoritmi di reinforcement learning già discussi nei capitoli precedenti. Per la creazione di questi ambienti sono state scelte ed utilizzate due tecnologie software: OpenAi Gym e pyBullet. OpenAi Gym, come già accennato, è un toolkit nato per la ricerca in Reinforcement Learning che, oltre ad includere una collezione in continua crescita di ambienti, fornisce interfacce ed astrazioni per la definizione di nuovi ambienti. PyBullet (capitolo 2) invece è un framework che mette a disposizione una motore grafico per la simulazione 3D di corpi solidi. Quindi gli ambienti sono creati utilizzando l'interfaccia OpenAi Gym mentre pyBullet ha il compito di gestire le dinamiche e la fisica dei componenti e dell'ambiente stesso. Vengono proposti quattro ambienti suddivisi per difficoltà: due ambienti più semplici e due con un livello di difficoltà abbastanza elevato.

3.1 Progettazione degli ambienti

Uno degli ambienti ricreati tratta un classico problema in letteratura in machine learning: il ***cart-pole problem***. Il funzionamento di questo problema è riassunto in figura 3.1. Il problema consiste nel controllare la posizione x di un carrello (cart) affinché il palo (pole) montato sopra di esso rimanga il più possibile in posizione verticale. Si vogliono realizzare due versioni di questo problema: la prima più semplice consiste nel connettere il palo alla base del carrello attraverso una connessione con solamente due gradi di libertà. In questo modo il palo potrà solamente cadere in avanti o indietro. La posizione del cart quindi potrà solamente spostarsi in avanti o indietro. La seconda versione cartPoleHard invece prevede che il pole sia connesso

alla superficie del cart con connessione a sei gradi di libertà. In questo modo il pole può cadere in qualsiasi direzione e pertanto il cart potrà muoversi in tutte e quattro le direzioni.

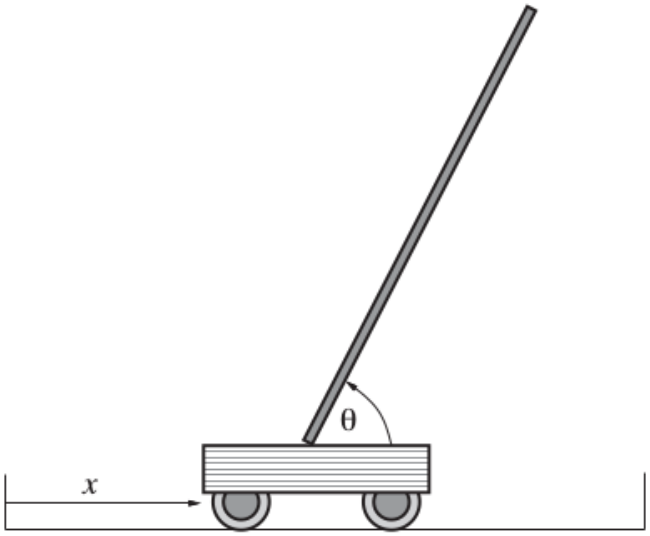


Figura 3.1: Setup del problema di bilanciamento di un palo sopra un carrello mobile.

Il ***MobilePlane problem*** prevede un piano quadrato rialzato sul quale è posta una pallina. Questo piano, muovendosi, porta la pallina a spostarsi e a mantenere quindi una posizione instabile. Si tratta anche questo di un problema di bilanciamento (balancing) in cui l'obiettivo è mantenere la pallina sul piano ed evitare che questa fuoriesca. Anche qui sono presenti due versioni del problema. Il problema "simple" prevede la possibilità del piano, e quindi della pallina, di muoversi solamente lungo due direzioni, mentre la versione più complessa comprende tutte e quattro le direzioni.

3.1.1 Caratteristiche

Come è già stato accennato precedentemente, gli ambienti devono essere di natura episodica per cui deve essere presente uno stato terminale. Ogni episodio deve terminare sia in caso di raggiungimento dell'obiettivo sia in caso contrario. Se l'obiettivo viene raggiunto parliamo di stato terminale di successo, mentre se viene raggiunto lo stato terminale senza conseguimento dell'obiettivo parliamo di stato terminale di insuccesso. Nei casi specifici quindi gli obiettivi degli ambienti qui proposti sono il raggiungimento di 500 step consecutivi (nello stesso episodio). Diamo quindi per raggiunto l'obiettivo di bilanciamento del pole ,o della pallina, se trascorre un certo

numero di step senza che il palo o la pallina cada. Il limite viene posto per evitare che gli episodi portino via troppo tempo in fase di learning.
Il reward utilizzato in questi ambienti è un reward cumulativo: ciò significa che ogni step di un determinato episodio ha come reward in uscita sempre 1. In questo modo viene sommato ai reward all'interno dello stesso episodio. Così facendo, più il reward è alto più il pole o la pallina vengono mantenuti stabili e bilanciati.
In questi ambienti dove il problema è quello di mantenere un certo stato il più a lungo possibile (il bilanciamento) un reward unico e cumulativo può essere una buona scelta. Non va però ignorato che, in linea generale, l'utilizzo di diversi valori di reward aiutano l'agente ad intraprendere scelte migliori e ad indirizzare l'agente stesso verso la scelta di azioni che lo guidino verso l'obiettivo finale.
Definito come deve essere assegnato il reward è opportuno definire di quali valori dovrà essere composto lo stato dell'ambiente. Lo stato è composto da un vettore di parametri che descrivono l'ambiente nella sua condizione corrente. Il vettore dello stato è diverso per ogni ambiente e varia anche in dimensionalità a seconda della difficoltà dell'ambiente stesso. Dalla tabella 3.1 è possibile osservare i vettori di stato per i vari ambienti. Nel *cart-pole problem* l'ambiente è descritto da 4 stati: 2 stati che descrivono il cart ovvero posizione (x) e velocità (x_dot) e 2 che descrivono il pole, ovvero angolo di inclinazione (theta) e velocità angolare (theta_dot).

Cart-pole Problem	
Parametro	Descrizione
X	Posizione lungo l'asse x del cart
X_dot	Velocità del cart
Theta	Angolo di inclinazione del pole
Theta_dot	Velocità angolare del pole

Tabella 3.1: Definizione dei parametri per lo stato in cart-pole problem versione base.

Nel *cart-pole problem* versione "hard" gli stati diventano 8: in questo caso gli stati sono i medesimi dell'ambiente semplice, ma comprendono anche i dati dell'asse y (tabella 3.2).

Cart-pole Problem Hard	
Parametro	Descrizione
X	Posizione lungo l'asse x del cart
X_dot	Velocità del cart lungo l'asse x
Theta	Angolo in x di inclinazione del pole
Theta_dot	Velocità angolare del pole lungo l'asse x
Y	Posizione lungo l'asse y del cart
Y_dot	Velocità del cart lungo l'asse y
Theta1	Angolo in y di inclinazione del pole
Theta1_dot	Velocità angolare del pole lungo l'asse y

Tabella 3.2: Definizione dei parametri per lo stato in cart-pole problem versione "Hard".

Le azioni che sono permesse in questi due ambienti sono rispettivamente due per l'ambiente semplice, che prevede lo spostamento a destra o a sinistra del cart lungo l'asse x, e quattro per la versione Hard che prevede quindi il movimento del cart nelle 4 direzioni.
Per quanto riguarda invece l'ambiente *MobilePlane* abbiamo rispettivamente 2 parametri di stato per l'ambiente semplice e 4 per l'ambiente "hard".

MobilePlane Hard Action table	
Azione	Descrizione
Decremento piano X	Decremento dell'angolo di inclinazione del piano lungo l'asse x
Incremento piano X	Incremento dell'angolo di inclinazione del piano lungo l'asse x
Decremento piano Y	Decremento dell'angolo di inclinazione del piano lungo l'asse y
Incremento piano Y	Incremento dell'angolo di inclinazione del piano lungo l'asse y

Tabella 3.3: Definizione delle azioni possibili per MobilePlane Hard

Il primo ambiente è caratterizzato solamente dalle componenti in x per quanto riguarda l'inclinazione del piano e posizione della pallina, mentre per la versione

più complessa si aggiungono a queste dimensioni anche i rispettivi valori per l'asse y. Le azioni in questi ambienti permettono semplicemente la rotazione del piano e sono descritti in tabella 3.3 (per la versisone semplice si considerino solo le prime due azioni presenti in tabella).

3.2 Struttura degli ambienti

Il capitolo 2 descrive il toolkit di sviluppo gym OpenAi e il motore grafico pyBullet utilizzati rispettivamente per creare e permettere la simulazione 3D degli ambienti sopra descritti. In questo paragrafo vengono esposti nel dettaglio tali ambienti in riferimento alla struttura base introdotta nel paragrafo 2.3.2.

Tutti gli ambienti utilizzano l'interfaccia comune messa a disposizione dal toolkit gym OpenAi che permette così l'utilizzo degli ambienti a blackbox, poichè tutti gli ambienti forniscono gli stessi metodi. I metodi forniti dagli ambienti, agli algoritmi di learning che poi li utilizzeranno, prendono il nome di `reset` e `step` .
Il metodo `reset` ha il compito di resettare l'ambiente, portandolo nel suo stato iniziale. Comprende le istruzioni per il loading di tutti gli elementi solidi che dovranno essere caricati dal motore grafico e ritornano lo stato iniziale dell'ambiente.
Il metodo `step` ha il compito di eseguire un azione e modificare di conseguenza lo stato dell'ambiente. Questo metodo coincide quindi con uno step di esecuzione dato dall'algoritmo di learning. Il metodo presenta al proprio interno un vettore contenente tutte le azioni possibili come mostrato nel codice a seguire.

```
deltav = [-1. * dv, 1. * dv][action]
```

`deltav` prende quindi il valore dell'azione scelta dalla variabile `action` passata come parametro al metodo `step`. La variabile `deltav` viene poi utilizzata dal metodo `setJointMotorControl2` (paragrafo 2.4.1) per modificare lo stato del corpo solido all'interno della simulazione.
L'interfaccia messa disposizione da Gym prevede anche delle variabili che foniscono informazioni dell'ambiente stesso. Vengono infatti utilizzate le variabili `action_space` e `observation_space` per definire rispettivamente il numero di azioni e la dimensionalità dello stato dell'ambiente. In questo modo gli algoritmi di learning possono utilizzare questi dati in maniera automatica (ad esempio definire il numero di neuroni di input in una rete neurale sulla base di `observation_space`).

3.2.1 Ambienti 3D

Grazie all'unione del toolkit OpenAi Gym con pyBullet è stato possibile realizzare gli ambienti descritti. OpenAi Gym fornisce un interfaccia che permette di trattare la simulazione degli ambienti in maniera episodica mentre Bullet fornisce gli elementi di fisicità tipici della natura. Il paragrafo 2.5.1 introduce i concetti di joint e link utilizzati per descrivere i corpi solidi attraverso URDF. In questo paragrafo vengono utilizzati questi concetti per illustrare il funzionamento di tali corpi nell'ambiente 3D.

Ambiente cart-pole

L'ambiente cart-pole viene caricato all'interno del metodo reset attraverso il comando `loadURDF`:

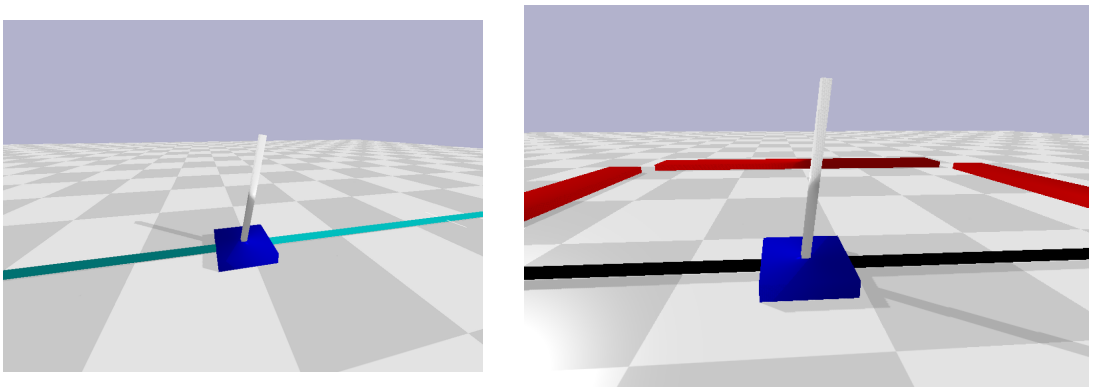
```
self.cartpole = p.loadURDF("cartpole.urdf", [0, 0, 0])
p.loadURDF("plane.urdf", [0, 0, -0.05])
```

L'ambiente è composto da 2 file URDF: plane.urdf che ha il solo scopo di definire l'appoggio ai corpi solidi della scena, e il file cartpole.urdf in cui è definito il cart. Il file URDF che compone il robot è formato da tre solidi, tre cuboidi che modellano il pole, il cart e uno slider sul quale il cart può scivolare. Sono presenti due elementi di joint. `slider_to_cart` che lega lo slider al cart attraverso la proprietà *prismatic* in modo che possa muoversi lungo l'asse X, mentre il joint `cart_to_pole` lega il cart al pole attraverso la proprietà *continuous* che permette al pole di poter cadere lungo l'asse Y.
L'ambiente cart-pole in versione hard è più complesso e prevede più file URDF da caricare nell'ambiente:

```
self.cartpole = p.loadURDF("cart-pole-hard.urdf", [0, 0, 0])
self.wallR = p.loadURDF("wall.urdf", [2.9, 0, 0])
self.wallT = p.loadURDF("wall.urdf", [0, 2.9, 0],
                            baseOrientation=[0, 0, 1, 1])
 self.wallB = p.loadURDF("wall.urdf", [0, -2.9, 0],
                            baseOrientation=[0, 0, 1, 1])
self.wallL = p.loadURDF("wall.urdf", [-2.9, 0, 0])
p.loadURDF("plane.urdf", [0, 0, -0.05])
```

```
...
p.resetJointState(self.cartpole, 2, angle1)
p.resetJointState(self.cartpole, 3, angle2)
```

In questa versione il cart e il pole sono caricati da due file URDF separti poichè nella versione hard non sono legati da nessun vincolo. Gli elementi wall.urdf invece sono solidi che rappresentano i muri che circondano la zona in cui il cart può muoversi (in rosso in figura 3.2b). `baseOrientation` definisce l'orientazione del muro affinché possa essere posizionato correttamente. Le variabili `angle1` e `angle2` sono valori randomici che determinano un certo grado di rumore iniziale per quanto riguarda l'orientazione del pole così che il pole non venga inizializzato già in posizione bilanciata.



(a) Versione semplice

(b) Versione Hard

Figura 3.2: Ambiente grafico Bullet Physics con le due versioni del cart-pole system

Il file "cart-pole-hard.urdf" è più complesso rispetto alla sua versione semplice. Alla versione hard infatti si aggiunge un cuboide (non visibile) che funge da collegamento tra il carrello e il palo per garantire il movimento in tutti e sei i gradi di libertà.

Ambiente MobilePlane

Questo ambiente è di più semplice realizzazione poichè consiste solamente di 2 file URDF (oltre a plane.urdf):

```
self.plane = p.loadURDF("plane-mobile.urdf", [0, 0, 0])
...
```

```
self.ball = p.loadURDF("ball.urdf", [ballPosX, ballPosY, 0.2])
p.loadURDF("plane.urdf", [0, 0, -0.05])
```

Anche qui sono presenti due variabili randomiche `ballPosX` e `ballPosY` per la generazione semi-casuale della posizione iniziale della pallina.
Il file "plane-mobile.urdf" è composto da quattro cuboidi. Il cuboide principale è il piano su cui la pallina deve rimanere bilanciata mentre gli altri tre cuboidi hanno solamente lo scopo di permettere il movimento del piano. La struttura gerarchica con cui sono collegati i cuboidi prevede il piano come ultimo elemento in fondo alla gerarchia. I cuboidi sono collegati tra loro attraverso diverse tipologie di joint. Il primo è connesso al secondo attraverso la proprietà *fixed* mentre i restanti due sono collegati a catena attraverso la proprietà *revolute* che permette il movimento lungo un unico asse. In questo modo un cuboide può muovere solamente l'asse x mentre l'altro cuboide può muovere solamente l'asse y mentre il cuboide in testa alla gerarchia mantiene tutta la struttura fissata con la proprietà *fixed*. In questo modo il piano che è collegato può muoversi e inclinarsi lungo entrambi gli assi.
Mentre la versione hard utilizza interamente la struttura, la versione semplice del problema sfrutta solamente il movimento di un unico asse.
Questa soluzione, adottata anche per l'ambiente cartpole, sostituisce la proprietà *floating* degli attributi di joint del linguaggio URDF che prevede il movimento lungo tutti e sei i gradi di libertà poiché non è ancora stata implementata in Bullet.

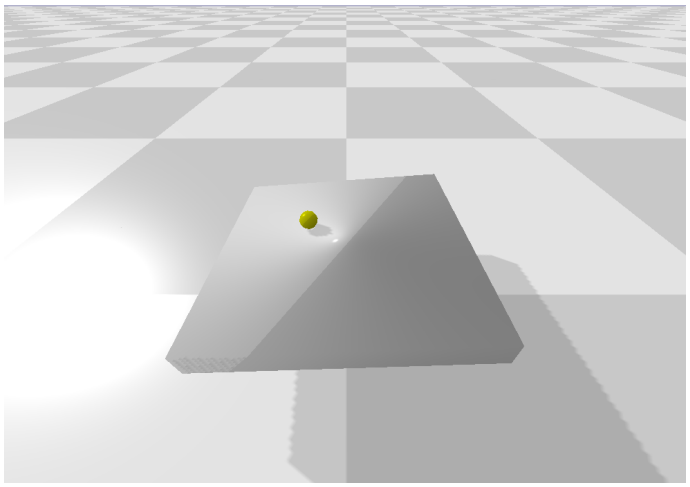


Figura 3.3: Ambiente MobilePlane

Definito nello specifico il legame tra i vari solidi che compongono l'ambiente, si vuole definire nel dettaglio il funzionamento del metodo `step`:

```
def step(self, action):

    # esegui azione

    ballPos, _ = p.getBasePositionAndOrientation(self.ball)

    self.state = (p.getJointState(self.plane, 0)[0],
       p.getJointState(self.plane, 1)[0], ballPos[0], ballPos[1])

    done = self._isDone(ballPos)
    reward = 1.0

    return np.array(self.state), reward, done, {}
```

Il funzionamento di come l'azione viene eseguita è già stato discusso precedentemente. Nel codice soprastante si vuole evidenziare come le componenti che formano lo stato vengono estratte dai dati. Il metodo `getJointState` permette di recuperare alcune variabili di stato di un determinato joint come posizione, velocità ed altri fattori. In questo caso si è interessati solamente alla posizione dei joint n° 0 e 1 (ovvero quelli che determinano il movimento del piano). Allo stato finale dell'agente viene poi aggiunta la posizione in x e in y date dall'array `ballPos` recuperata dal metodo `getBasePositionAndOrientation`. Il return del metodo `step` è composto da diverse variabili. Oltre al vettore dello stato dell'ambiente, il metodo ritorna il valore di reward (posto sempre a 1 come spiegato precedentemente) e un flag `done`. Il flag `done` indica quando l'ambiente deve essere resettato poiché è stato raggiunto uno stato non valido che nell'ambiente in questione consiste nella caduta della pallina al di fuori del piano. L'ultimo parametro ritornato è invece a valore nullo poichè non è utilizzato; tipicamente è utilizzato come variabile contenente informazioni utili dell'environment che è stata mantenuto nel codice poichè fornita dall'interfaccia di OpenAi Gym.

Capitolo 4

Implementazione degli algoritmi di learning

In questo capitolo vengono presentati e descritti gli algoritmi di learning utilizzati per l'apprendimento degli ambienti descritti precedentemente nella teoria presente nel capitolo 1. Verranno quindi illustrate nel dettaglio le implementazioni, con riferimenti al codice, degli algoritmi di Q-Learning e DQN. Q-Learning con utilizzo della forma tabellare prevede una Q-table in cui vengono rappresentate le coppie stato-azione. A causa della natura di questo algoritmo, lo stato dell'ambiente viene discretizzato in bucket e per questo motivo negli ambienti più complessi può non ottenere risultati buoni. Il secondo metodo, DQN (Deep Q-Network) è un algoritmo più avanzato che prevede l'utilizzo di una rete neurale al posto della Q-Table. La rete neurale, per gli ambienti più complessi, è costruita su più livelli nascosti e questo permette prestazioni migliori, tenendo in considerazione che questo metodo non necessita di discretizzazione dello stato. Con questa tecnica non viene valutata ogni possibile coppia stato-azione ma il learning consiste nello stimare il comportamento della funzione valore mediante modifica dei pesi interni alla rete neurale.
Grazie all'interfaccia comune ai vari ambienti realizzati e alla loro natura episodica, gli algoritmi, anche se diversi, mantengono le stesse modalità di interazione con l'ambiente.

```
for episode in range(self.num_episodes):
    state = self.env.reset()
    for t in range(self.max_t):
        self.env.render()
        action = self.choose_action(state_0)
        state2, reward, done, _ = self.env.step(action)
        # Update della Q-Table o rete neurale
        state = state2

        if done:
            break
```

Lo pseudo-codice sopra riportato mostra la struttura su cui sono basati gli algoritmi di learning implementati. La natura degli algoritmi prevede un ciclo iterativo ad episodi nel quale l'agente dovrà apprendere. Ad ogni inizio episodio il metodo `reset` provvede a resettare l'ambiente restituendo lo stato iniziale. All'interno di ogni episodio è presente un altro ciclo che itera gli step t di simulazione dell'ambiente. Il metodo `render` attua la simulazione a schermo mentre il metodo `choose_action` sceglie un'azione sulla base dello stato iniziale che viene poi passata come parametro al metodo `step` dell'environment che esegue lo step di simulazione sull'ambiente. Successivamente le variabili `state state2 reward action` vengono utilizzate per il passo di learning vero e proprio, che dipende dall'algoritmo utilizzato. `step` oltre a restituire il nuovo stato `state2`, restituisce anche il flag `done` utile a comprendere se occorre resettare l'ambiente e passare all'episodio successivo, oppure continuare al prossimo step di simulazione.

4.1 Algoritmo Q-Learning

Come già illustrato nel capitolo introduttivo, Q-Learning è un metodo temporal difference off-policy per il reinforcement learning, ovvero aggiorna la funzione valore ad ogni passo di step. L'approccio off-policy indica invece un procedimento che utilizza due tipi di policy: una policy totalmente greedy che viene utilizzata per l'esplorazione (*behavior policy*) delle coppie stato-azione in modo tale da non ricadere in ottimi locali, e una policy di apprendimento vera e propria con politica ϵ-greedy chiamata anche *target policy*. Attraverso un fattore di esplorazione (exploration rate) si spinge l'algoritmo verso una scelta esplorativa piuttosto che di apprendimento. Con l'avanzare degli episodi l'exploration rate viene decrementato per dare più peso alla policy di apprendimento. Nella pratica quindi si cerca di mantenere una politica maggiormente esplorativa all'inizio del training per poi spostare l'attenzione sul fattore di apprendimento.

La natura off-policy dell'algoritmo richiede in ingresso parametri aggiuntivi oltre a quelli forniti dall'ambiente come dimensionalità dello stato o numero di azioni eseguibili. Tre sono i parametri aggiuntivi: α, ϵ, γ che rappresentano rispettivamente learning rate, exploration rate e discounting rate. I fattori di learning e discount sono rappresentati nella formula 1.13 dell'algoritmo di Q-Learning e vengono calcolati da funzioni che prendono in input il time step t:

```
def _get_explore_rate(self, t):
    return min(self.max_exp_rate,self.min_exp_rate +
        (1 - self.min_exp_rate) * math.exp(-self.decay_speed * t))

def _get_learning_rate(self, t):
    return max(self.min_learn_rate,
        min(0.5, 1.0 - math.log10((t + 1) / 100.0)))
```

`max_exp_rate` è il fattore massimo da cui parte l'algoritmo al time step 0 e viene fissato a 0.8. Ciò significa che inizialmente la politica di esplorazione verrà scelta con percentuale del 80% rispetto alla politica di learning. La formula descritta dal codice permette il decremento del fattore di esplorazione e la velocità di tale decremento è data dalla variabile `decay_speed` che viene richiesta come parametro di ingresso. Più `decay_speed` assume valore prossimo allo zero, più lento sarà il

decremento di ϵ. `decay_speed` assume valori diversi a seconda dell'ambiente scelto: se l'ambiente è semplice e permette un learning veloce, allora la velocità di decadimento dovrà essere più elevata, mentre per ambienti complessi la velocità di decadimento dell'esplorazione dovrà essere più lenta in modo da permettere sempre un giusto compromesso tra esplorazione e apprendimento.
Anche il fattore α (learning rate) decrementa all'aumentare del time step: il massimo valore che può assumere è fissato a 0.5 mentre il valore minimo a 0.01 . Il learning rate, o tasso di apprendimento, determina con quale estensione le nuove informazioni acquisite sovrascriveranno le vecchie informazioni. Un fattore 0 impedirebbe all'agente di apprendere, al contrario un fattore pari ad 1 farebbe sì che l'agente si interessi solo delle informazioni recenti. Risultati sperimentali dimostrano che il learning rate dinamico utilizzato in questo algoritmo da risultati migliori rispetto ad un tasso di apprendimento fisso.
L'unico fattore fisso è il discount rate. Il fattore γ determina l'importanza delle ricompense future. Un fattore pari a 0 renderà l'agente "opportunista" facendo sì che consideri solo le ricompense attuali, mentre un fattore tendente ad 1 renderà l'agente attento anche alle ricompense che riceverà in un futuro a lungo termine. In letteratura è comune utilizzare discount rate prossimo ad 1 perciò è stato scelto $\gamma = 0.99$.

```
def choose_action(self, state, explore_rate):
    # Select a random action
    if random.random() < explore_rate:
        action = random.randint(0, self.num_actions-1)
    # Select the action with the highest q
    else:
        action = np.argmax(self.q_table[state])
    return action
```

Il metodo `choose_action` permette la scelta dell'azione da intraprendere sulla base del trade-off illustrato precedentemente tra esplorazione e apprendimento. Si va infatti a scegliere la beauvoir policy greedy che consiste in una scelta randomica dell'azione, oppure viene utilizzata la target policy che preleva l'azione con il Q-value maggiore dalla tabella stato-azione.

```
best_q = np.amax(self.q_table[state])
self.q_table[state_0 + (action,)] += learning_rate * (
    reward + gamma * (best_q) - self.q_table[state_0 + (action,)])
```

L'update della Q-Table segue la formula 1.13 dove `best_q` è il valore maggiore ottenuto per lo stato `state` e `gamma` è il discount rate.

Per utilizzare la tabella Q-table per la memorizzazione dei valori per le coppie stato-azione è necessario discretizzare gli stati. La discretizzazione è stata realizzata attraverso una tecnica a bucket. Fissato il numero di bucket per ogni parametro dello stato si sfrutta una funzione che, preso in input lo stato in forma non discretizzata, permette lo "smistamento" e quindi la discretizzazione restituendo il bucket a cui viene assegnato.

```
def state_to_bucket(self, state):
    bucket_indice = []
    for i in range(len(state)):
        if state[i] <= self.sBounds[i][0]:
            bucket_index = 0
        elif state[i] >= self.sBounds[i][1]:
            bucket_index = self.bucket_size[i] - 1
        else:
            # Mapping the state bounds to the bucket array
            bound_width = self.sBounds[i][1] - self.sBounds[i][0]
            offset = (self.bucket_size[i] - 1) *
                       self.sBounds[i][0] / bound_width
            scaling = (self.bucket_size[i] - 1) / bound_width
            bucket_index = int(round(scaling * state[i] - offset))
        bucket_indice.append(bucket_index)
    return tuple(bucket_indice)
```

La Q-Table è inizializzata come segue

```
self.q_table = np.zeros(bucket_size + (self.num_actions,))
```

dove `bucket_size` è una tupla di n elementi dove n è la dimensionalità dello stato mentre `num_actions` è la dimensionalità data dal numero delle azioni ottenendo quindi la forma tabellare con coppie stato-azione.
Tutte le configurazioni dei vari parametri compreso il numero di bucket scelto per ogni stato verrà discusso assieme ai risultati ottenuti nel capitolo seguente.

4.2 Algoritmo Deep Q-Network

La soluzione al problema di discretizzazione degli stati con conseguente perdita di informazioni che potrebbero risultare importanti e decisive al learning dell'ambiente viene risolta grazie all'algoritmo di Deep Q-Nework. La Q-Table infatti viene sostituita da una rete neurale la quale non necessita più di alcuna forma di discretizzazione che approssima la funzione valore. La rete prende lo stato come input e produce una stima della funzione valore per ogni azione. Il nome Deep Q-Network deriva dall'utilizzo di più livelli nascosti nella composizione della rete.
Purtroppo l'utilizzo di una rete neurale per rappresentare la funzione valore non sempre produce risultati stabili. Per rispondere a questo problema sono stati introdotti quindi alcune idee chiave per stabilizzare il training.
Il primo concetto da illustrare è quello dell'*experience replay*. Il problema con gli algoritmi di learning è quello di avere un alto livello di correlazione tra esperienze successive. Per questo motivo, la rete potrebbe facilmente andare in overfitting con conseguente fallimento nel creare un apprendimento generalizzato e quindi più genuino. L'idea attraverso l'experience replay è quindi quella di salvare le esperienze in una memoria chiamata *replay memory* e durante ogni passo di learning recuperare in maniera randomica un campione di tali transazioni. Similmente a quanto detto per Q-Learning, anche DQN utilizza metodologia off-policy per bilanciare la componente esplorativa con quella di apprendimento.

Oltre ai parametri richiesti in input per il funzionamento dell'agente già citati nell'algoritmo Q-Learning come dimensionalità dello stato, numero di azioni, `decay_speed` (4.1) vengono richiesti altri parametri riassunti nella tabella seguente.

Parametro	valore	Descrizione
num_input	/	Numero di neuroni per l'input layer (valore fornito dall'ambiente).
num_output	/	Numero di neuroni per l'output layer (valore fornito dall'ambiente).
num_hidden	64	Numero di neuroni per ogni hidden layer.
MEM_CAPACITY	100000	Capacità della replay memory.
learning rate	0.00025	Learning rate per l'optimaizer RMSprop.

Tabella 4.1: Parametri per la rete neurale

Le configurazioni mostrate in tabella sono quelle di default. Nel capitolo finale verranno discusse le varie configurazioni per ogni ambiente in modo da rendere più performante l'algoritmo sull'ambiente stesso, pertanto queste configurazioni sono solo indicative.

Per la realizzazione della rete neurale viene utilizzata la libreria python TensorFlow [8], framework opensource per il machine learning. In particolare della libreria TensorFlow si utilizzano le API fornite da Keras [5] per la creazione della rete neurale. Keras permette con poche righe di codice la creazione di una rete neurale in maniera molto intuitiva:

```
def createNet(self):
    net = Sequential()
    net.add(Dense(self.num_hidden, activation='relu',
                  input_dim=self.num_inputs))
    net.add(Dense(self.num_hidden, activation='relu',
                  input_dim=self.num_hidden))
    net.add(Dense(self.num_hidden, activation='relu',
                  input_dim=self.num_hidden))
    net.add(Dense(self.num_outputs, activation='linear'))

    optimizer = RMSprop(lr=self.lr)
    net.compile(loss='mse', optimizer=optimizer)

     return net
```

Il metodo `create_net` sopra citato permette la creazione di una rete neurale con tre livelli nascosti con funzione di attivazione ReLu (capitolo 1.2). Il livello finale consiste invece di un numero di neuroni pari al numero di azioni dell'environment con funzione di attivazione lineare. Il metodo `add` permette di aggiungere sequenzialmente i layer alla rete. Questi layer sono di tipologia "Dense" ovvero i neuroni sono *fully connected* con i neuroni del livello successivo. Invece di una semplice funzione di gradient descent, viene utilizzato un algoritmo più sofisticato per l'ottimizzazione ovvero RMSprop [12] e l'errore quadratico medio (mse) come funzione da minimizzare.
Poichè la rete implementa una tecnica di regressione (e non di classificazione) viene utilizzata come loss function l'errore quadratico medio invece di cross entropy. Gli errori calcolati dalla loss function saranno propagati all'indietro nella rete mediante un passo backward, seguendo la logica di discesa del gradiente con l'intento di minimizzare l'errore.

Una volta creata la rete viene lanciato il metodo `Train` che contiene la logica di governo dell'environment:

```
for iteration in range(num_iterations):
    state = self.env.reset()
    R = 0
    for step in range(max_game_steps):
        self.env.render()
        action = self.choose_action(state)
        state2, r, done, _ = self.env.step(action)
        if done: # terminal state
            state2 = None

        obs = (state, action, r, state2)
        self.memory.add(obs)

        self.epsilon = self.MIN_EPSILON + (self.MAX_EPSILON
        - self.MIN_EPSILON) * math.exp(-self.LAMBDA * iteration)

        self.updateNet(discount_rate, batch_size)

        state = state2
        R += r

        if done:
            break

    print("iteration {} Total reward: {} ".format(iteration,R))
```

La logica di funzionamento rispecchia quanto detto ad inizio capitolo. Il metodo `choose_action` funziona similmente al metodo introdotto in Q-Learning: l'azione viene scelta sulla base di una policy esplorativa e una policy di apprendimento. L'environment esegue l'azione è ritorna un nuovo stato ed un reward. L'agente quindi prende in osservazione la nuova tupla $(s,\ a,\ r,\ s')$, la aggiunge in replay memory e aggiorna ϵ, variabile che controlla la velocità di decadimento e permette alla policy esplorativa di diminuire di efficacia in funzione della policy di appren-

dimento. Come ultimo step viene aggiornata la rete e quindi eseguito il passo di learning vero e proprio.

```
def updateNet(self, discount_rate, batch_size):

    batch = self.memory.pick_n(batch_size)
    batchLen = len(batch)
    no_state = numpy.zeros(self.num_inputs)
    states = numpy.array([o[0] for o in batch])
    states_ = numpy.array([(no_state if o[3] is None
                      else o[3]) for o in batch])
    p = self.predict(states)
    p_ = self.predict(states_)
    x = numpy.zeros((batchLen, self.num_inputs))
    y = numpy.zeros((batchLen, self.num_outputs))

    for i in range(batchLen):
        o = batch[i]
        s, a, r, s_ = o
        t = p[i]
        if s_ is None:
            t[a] = r
        else:
            t[a] = r + discount_rate * numpy.amax(p_[i])
        x[i] = s
        y[i] = t
    self.net.fit(x, y, batch_size=batch_size, epochs=1, verbose=0)
```

Nel codice sopra riportato è descritto il metodo per l'aggiornamento dei pesi nella rete neurale. Il metodo prende come parametri di input il fattore di discount (settato a 0.99 come per Q-Learning) e la dimensione di batch da considerare. Vengono quindi selezionati randomicamente dalla replay memory n elementi che comporranno il nostro batch. Vengono quindi fatte le predizioni per tutti gli stati iniziali e finali del batch in un unico step. La variabile `no_state` viene utilizzata come variabile dummy di stato quando lo stato finale è *None*, questo perchè Keras non supporta lo stato *None* che in questo caso viene sostituito con un array di zero. La

variabile `p` ora contiene le predizioni per gli stati iniziali è verrà usata come variabile target per il learning (variabile T nell'algoritmo 3).
La variabile `p_` invece contiene le predizioni degli stati finali e viene utilizzata nella parte finale della formula $\max_a Q(s', a)$ (algoritmo 3). Per ogni campione prelevato dalla replay memory vengono salvate iterativamente le variabili target e stato in due vettori x e y che vengono poi dati in input alla funzione `fit` che esegue uno step di gradient descend aggiornando la rete.

Per completezza di seguito viene mostrato il codice della classe Memory che implementa la replay memory:

```
class Memory():
    def __init__(self, capacity):
        self.memo = []
        self.capacity = capacity

    def add(self, sample):
        self.memo.append(sample)

        if len(self.memo) > self.capacity:
            self.memo.pop(0)

    def pick_n(self, n):
        n = min(n, len(self.memo))
        return random.sample(self.memo, n)
```

È costituita di 2 metodi: il metodo `add` semplicemente aggiunge il campione alla memoria, mentre il metodo `pick_n` ritorna un numero n di campioni presi dalla memoria. Se la memoria risulta piena, `add` provvede ad eliminare il primo campione aggiunto per far posto al nuovo.

Capitolo 5

Configurazioni e risultati

Nel seguito di questo capitolo conclusivo vengono proposte le configurazioni e i risultati ottenuti dai metodi Q-Learning e Deep Q-Network applicati ad ognuno degli ambienti realizzati. Le configurazioni verteranno sui principali parametri già discussi nei precedenti capitoli che subiranno leggere modifiche a seconda dell'ambiente eseguito.
Per DQN i parametri configurabili sono learning rate, ϵ (exploration rate), velocità di decadimento di ϵ, numero di livelli nascosti, numero di neuroni per ogni livello nascosto, mentre gli unici valori che non subiranno cambiamenti sono il discount rate (0.99), la capacità della replay memory (100000) e la dimensione dei mini-batch (64) utilizzati per estrarre le tuple dalla reply memory.
Per quanto riguarda invece Q-Learning, i parametri configurabili sono sempre learning rate, ϵ e velocità di decadimento, più i parametri caratteristici dell'algoritmo quali le dimensionalità dei bucket per la Q-Table.
Ogni episodio di training ha durata massima 700 step, posto come limite per non allungare ulteriormente i tempi di learning, mentre il numero di step minimo per considerare l'avvenuto learning di bilanciamento è di 500 step. L'episodio quindi termina prima se il bilanciamento fallisce, altrimenti si considera l'oggetto bilanciato se l'episodio dell'ambiente si protrae fino al 500esimo step o oltre.
Il completamento della fase di training prevede il salvataggio della Q-Table per Q-Learning e dei pesi della rete neurale per DQN. Questi risultati vengono sottoposti a testing su 500 episodi per ricavare l'indice di accuratezza delle coppie agenti-ambienti.
Nel seguito del capitolo per ogni configurazione realizzata, vengono riportati e discussi i grafici ottenuti dagli algoritmi Q-Learning e Deep Q-Network. Infine vengono confrontati i due metodi, riepilogando i valori di performance ottenuti.

5.1 Configurazione e risultati in ambiente cartPole semplice

L'ambiente cartPole semplice è costituito da un carrello posto su un binario che permette il movimento in due sole direzioni. La configurazione adottata per questo ambiente è riassunta in tabella 5.1 :

Deep Q-Network		Q-Learning	
Parametro	valore	Parametro	valore
learning rate	0.00025	learning rate	0.5 - 0.1
decay speed	0.001	decay speed	0.0001
hidden layers	1	bucket size	(5, 5, 15, 5)
ϵ	0.9	ϵ	0.8
hidden neurons	64		

Tabella 5.1: Configurazione parametri in ambiente cartPole semplice.

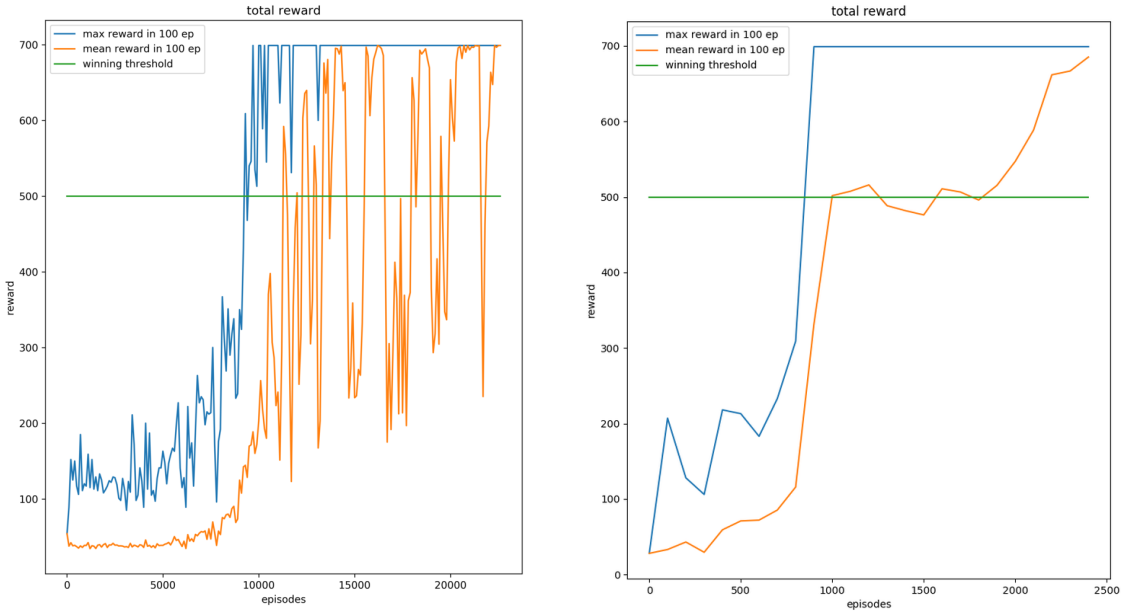
Q-Learning

Il parametro più complesso da settare è la dimensionalità dei bucket per ogni stato dell'ambiente. Dopo varie configurazioni provate, quella mostrata in tabella risulta essere la più performante per lo scopo prefissato (raggiungere il bilanciamento nel numero minore possibile di episodi). La velocità di decadimento fissata a 0.0001 permette ad ϵ un'esplorazione decrescente fino ad arrivare agli ultimi episodi di training dove il valore è prossimo a 0.01.
Il grafico 5.1a mostra le performance del metodo Q-Learning in fase di training per l'ambiente cartPole semplice. Il grafico riporta l'andamento del total reward medio ogni 100 episodi (arancione) e il total reward massimo ogni 100 episodi. L'algoritmo si considera risolto quando completa con successo 500 episodi consecutivi: si nota infatti come nel grafico gli ultimi total reward registrati siano tutti sopra la soglia di vincita dell'algoritmo (linea verde).

Deep Q-Network

L'obiettivo non consiste solamente nel raggiungere il reward più alto, ma consiste nel farlo nel minor numero di episodi possibile. Per questo motivo e per la natura semplice dell'ambiente, è stato possibile configurare la rete con un unico livello nascosto anzichè tre come proposto nel precedente capitolo. Anche la decay_speed

è stata ridotta per permettere un decadimento più veloce del parametro ϵ poichè vogliamo ottenere il bilanciamento in un numero ridotto di episodi.
Nel grafico 5.1b viene mostrato il risultato del learning. La differenza di episodi rispetto a Q-Learning è evidente: con agente DQN, l'ambiente impara a bilanciare il pole in circa 2500 episodi, al contrario di Q-Learning dove ve ne sono richiesti più di 20mila. Si può notare come l'algoritmo DQN non raggiunga per tutti gli episodi il valore massimo stabilito dall'algoritmo di 700 step. Probabilmente ciò è dovuto alla terminazione dell'algoritmo che necessita solamente di 500 episodi consecutivi sopra i 500 step per episodio (soglia fissata dall'algoritmo).



(a) Q-Leaning

(b) Deep Q-Network

Figura 5.1: Grafici delle performance per l'ambiente cartPole semplice.

5.2 Configurazione e risultati in ambiente cartPole Hard

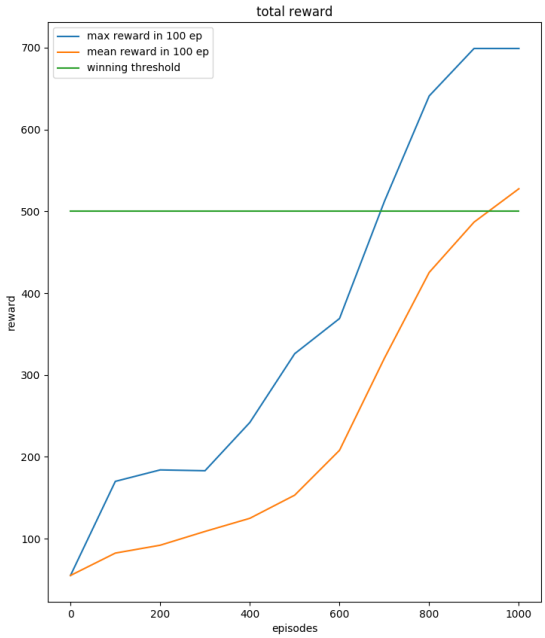
L'ambiente cartPole in versione hard prevede la possibilità del palo di poter cadere in qualsiasi direzione rendendo il training decisamente complesso. La complessità del training è da attribuire alla dimensionalità dello stato, composto appunto da 8 valori. L'algoritmo Q-Learning, basato su Q-Table che prevede la discretizzazione degli stati e la memorizzazione delle coppie stato-azione richiede una quantità di memoria superiore a quella disponibile dall'elaboratore su cui sono stati effettutati i test (12 GB). La dimensione dei *bucket_size* ridotta al minimo per permettere il training dell'ambiente non porta risultati concreti e rilevanti pertanto si è scelto di non riportare tali dati. In tabella 5.2 sono riportati solamente i parametri per DQN.

Deep Q-Network	
Parametro	valore
learning rate	0.00025
decay speed	0.001
hidden layers	3
ϵ	0.9
hidden neurons	64

Tabella 5.2: Configurazione parametri in ambiente cartPole hard.

Deep Q-Network

A differenza della sua variante semplice, la rete neurale è composta da tre livelli nascosti composti ciascuno da 64 neuroni. Il fattore di esplorazione parte da un valore molto alto (0.9) per favorire la capacità esplorativa poichè la dimensionalità dello stato è molto elevata. L'algoritmo si comporta abbastanza bene e in 1000 episodi circa la media dei reward su 100 episodi consecutivi supera il valore di 500. Il risultato riscontrato è buono anche se l'algoritmo è stato terminato prima della sua naturale conclusione. Questo perchè l'algoritmo realizzato tende ad evere comportamenti instabili. Ciò nonostante le perfomance per questo ambiente rimangono comunque alte (tabella 5.5). Nelle conclusioni sono esposti alcuni miglioramenti che possono essere implementati sull'algoritmo DQN utilizzato per rendere il learning di questo ambiente ancora migliore.



(a) Deep Q-Network

Figura 5.2: Grafico delle performance per l'ambiente cartPole hard.

5.3 Configurazione e risultati in ambiente mobilePlane semplice

L'ambiente mobile plane semplice è costituito da un piano inclinabile su cui poggia una pallina. In questa configurazione dell'ambiente il piano può inclinarsi solamente in 2 direzioni. Nella tabella 5.3 sono indicate le configurazioni adottate sia per entrambi gli agenti.

Deep Q-Network		Q-Learning	
Parametro	valore	Parametro	valore
learning rate	0.0025	learning rate	0.5 - 0.1
decay speed	0.001	decay speed	0.0001
hidden layers	1	bucket size	(25, 25)
ϵ	0.2	ϵ	0.4
hidden neurons	16		

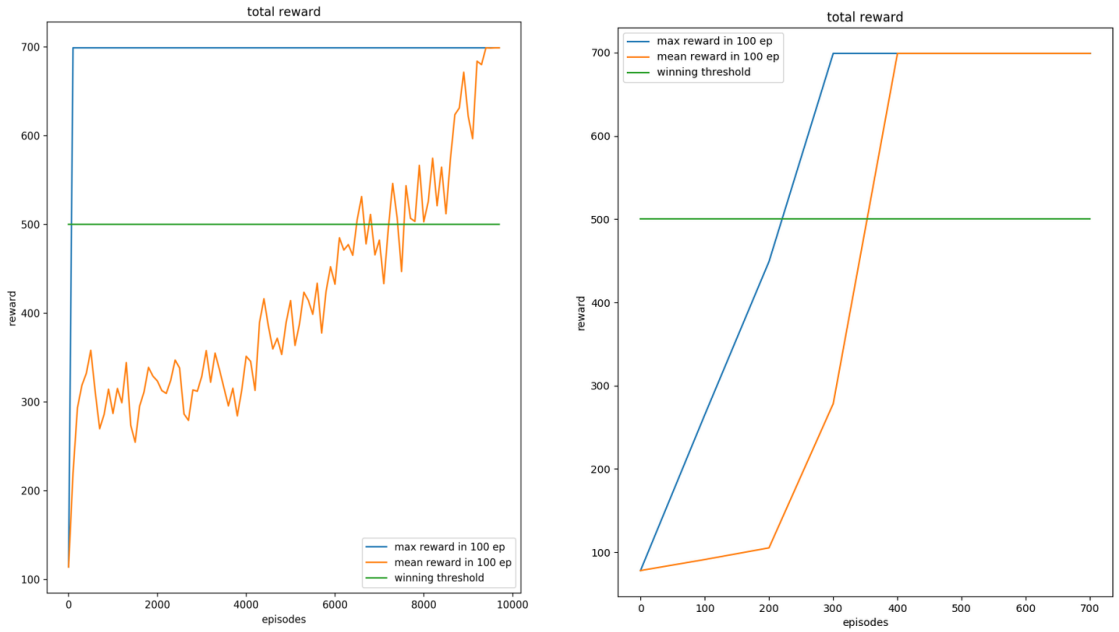
Tabella 5.3: Configurazione parametri in ambiente mobilePlane semplice.

Q-Learning

Con agente Q-Learning l'apprendimento è abbastanza lineare. Come si nota nel grafico 5.3a, l'agente conclude l'apprendimento in circa 10mila episodi. Lo stato è composto da una direzione che indica l'angolo di inclinazione del piano (valore positivo indica inclinazione in un verso mentre valore negaivo indica inclinazione nel verso opposto) e da uno stato che indica la posizione in x della pallina. Sono state impiegate altre configurazioni di *bucket size* maggiori ma con risultati minori.

Deep Q-Network

L'andamento del training con agente DQN invece è molto più lineare e veloce (grafico 5.3b). Data la natura semplice dell'ambiente, la rete neurale è configurata con un unico livello nascosto composto da 16 neuroni. Questa configurazione, unita ad una velocità di decadimento alta e ad un fattore di esplorazione limitato, permette all'agente di completare il processo di learning in meno di 800 episodi.



(a) Q-Leaning

(b) Deep Q-Network

Figura 5.3: Grafici delle performance per l'ambiente mobilePlane semplice.

5.4 Configurazione e risultati in ambiente mobilePlane hard

In questa versione del'ambiente mobilePlane, il piano si può inclinare in tutte e 4 le direzioni. È quindi una versione più complessa rispetto a quella vista precedentemente. Lo stato pertanto si arricchisce, rispetto alla versione semplice, di due nuove dimensionalità ovvero dell'inclinazione sull'asse y, e della posizione in y della pallina.
In tabella 5.4 vengono riassunti i parametri configurati per l'ambiente.

Deep Q-Network		Q-Learning	
Parametro	valore	Parametro	valore
learning rate	0.00025	learning rate	0.5 - 0.1
decay speed	0.0001	decay speed	0.0001
hidden layers	3	bucket size	(25, 25, 10, 10)
ϵ	0.2	ϵ	0.6
hidden neurons	64		

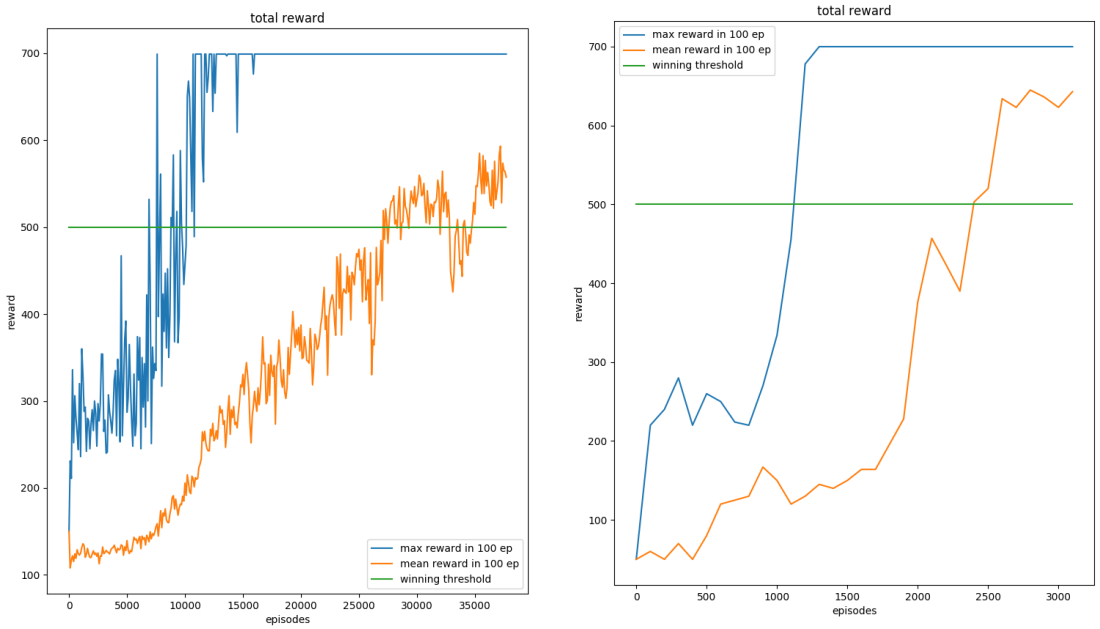
Tabella 5.4: Configurazione parametri in ambiente mobilePlane hard.

Q-Learning

In questo ambiente il learning risulta più lungo e meno lineare rispetto alla versione semplice. L'agente infatti necessita di più di 35mila episodi per raggiungere l'obiettivo di bilanciamento. Oltre alle difficoltà portate dall'ambiente, le tempistiche più lunghe possono essere attribuite al fattore di esplorazione, che in questo environment parte da 0.6, valore più alto rispetto alla versione semplice. La dimensionalità dei bucket viene mantenuta uguale per le dimensioni relative alle inclinazioni del piano (25) mentre le dimensionalità a valore 10 indicano le posizioni, in x e y, della pallina.

Deep Q-Network

L'algoritmo DQN, in linea con gli altri ambienti, si dimostra essere più performante rispetto a Q-Learning. Il parametro ϵ molto basso, unito ad una velocità di decadimento lenta permettono un learning stabile e lineare. Il training si conclude in circa 2100 episodi.



(a) Q-Leaning

(b) Deep Q-Network

Figura 5.4: Grafici delle performance per l'ambiente mobilePlane hard.

5.5 Confronto dei risultati

La misura di prestazione delle perfomance degli agenti sono calcolate attraverso l'indice di accuratezza:

$$accuracy = \frac{episodi\ completati\ correttamente}{episodi\ totali} \tag{5.1}$$

Dove si considerano gli episodi correttamente classificati quelli che superano la soglia impostata di 500 step consecutivi. Il numero totale di episodi considerati per calcolare questo indice è di 500. Il testing viene fatto utilizzando, per Deep Q-Network, i pesi salvati in fase di learning delle reti neurali, e le Q-Table salvate per l'algoritmo Q-Learning. Nella tabella seguente sono riportati i vari risultati per ogni ambiente e algoritmo adottato.

Ambiente	Agente	Accuracy	avg reward
cartPole simple	Q-Learning	100%	700.0
cartPole simple	DQN	99.60%	697.262
cartPole hard	DQN	87.20%	624.176
mobilePlane simple	Q-Learning	100%	700.0
mobilePlane simple	DQN	100%	700.0
mobilePlane hard	Q-Learning	68%	527.526
mobilePlane hard	DQN	95.20%	680.11

Tabella 5.5: Risultati ottenuti messi a confronto.

Mettendo a confronto i due algoritmi adottati si può notare che l'algoritmo Q-Learning ha risultati eccellenti nella maggior parte dei casi. Nonostante il passo di discretizzazione effettuato sugli stati, questo algoritmo ha un comportamento eccellente nelle versioni più semplici degli ambienti proposti. Le difficoltà più grandi di questo agente sono da riscontrare quando applicato alle versioni complesse: in ambiente mobilePlane Hard otteniamo una performance del 68% che produce comunque un reward medio superiore alla soglia minima di 500 step mentre le limitazioni di Q-Learning sono pienamente osservabili in ambiente cartPole hard. La numerosità elevata dello stato (8 valori) non ha permesso all'algoritmo di funzionare poiché la memoria fisica non è risultata sufficiente alla memorizzazione della Q-Table. È stato possibile utilizzare una Q-Table con numero di bucket per stato molto ridotto ma questa non ha permesso un training adeguato.
Gli ambienti in versione Hard hanno mostrato i limiti di Q-Learning ma anche le potenzialità di Deep-QNetwork: le reti neurali create per questi algoritmi si sono dimostrate molto performanti, sia in termini di risultati ottenuti, sia di memoria utilizzata.

Conclusioni e sviluppi futuri

Grazie ai risultati ottenuti sperimentalmente con l'utilizzo degli algoritmi Q-Learning e Deep Q-Network si è dimostrato come mediante l'addestramento tramite rinforzo (reinforcement learning) un agente autonomo sia in grado di apprendere automaticamente come bilanciare diversi oggetti (palo e pallina) in diverse situazioni e modalità. Sia Q-Learning con materializzazione della tabella di coppie stato-azione, che DQN con utilizzo di una rete neurale, ottengono ottimi risultati di accuratezza. DQN si è dimostrato migliore nel risolvere i task presentati in ambienti più complessi e di portare a compimento la fase di training in un numero di episodi considerevolmente inferiori.
L'utilizzo del toolkit openAI Gym mediante l'interfaccia fornita, ha permesso di realizzare ambienti standard che possono facilmente essere estesi in numerosità e quindi essere testati dagli algoritmi senza la modifica di quest'ultimi.
La simulazione 3D fornita dalla libreria pyBullet ha consentito un ambiente di sviluppo per la simulazione pulito ed elegante e ha permesso l'utilizzo semplice di strumenti di fisicità importanti come collisioni, moto dei corpi e gravità.
I risultati ottenuti in ambiente cartPole Hard hanno portato alla luce problematiche legate all'algoritmo DQN che risulta instabile in alcuni passi di training. Uno sviluppo futuro del progetto potrebbe essere legato quindi al miglioramento di tale algoritmo. Alcune migliorie semplici consistono nel cercare di diminuire il numero degli stati o utilizzare reti neurali di dimensioni minori. A tale proposito una variante potrebbe essere data da una diversa rappresentazione degli stati dell'ambiente: lo stato è calcolato tramite immagini che catturano la scena e che vengono poi mappate in matrici RGB dei pixel. In questo modo si può immaginare la costruzione di ambienti reali sui quali operare anziché simularli solamente. In letteratura esistono già alcune tecniche più complesse che potrebbero portare miglioramenti alla stabilità dell'algoritmo. Un esempio è dato in [13] con l'adozione della *prioritized experience replay memory*. L'experience replay consente agli agenti di RL di ricordare e riutilizzare esperienze del passato. L'experience replay adottata

in DQN viene campionata in maniera uniforme dalla replay memory mentre questo nuovo approccio cerca di dare priorità a quelle azioni considerate più importanti.

Bibliografia

[1] Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning: An Introduction*", The MIT Press Cambridge, Massachusetts

[2] C. J. C. H. Watkins, *Learning from delayed rewards*. PhD thesis, King's College, London, 1989.

[3] Anil K Jain, Jianchang Mao, and KM Mohiuddin, "*Artificial neural networks: A tutorial*". In *Computer* 3 (1996).

[4] V Mnih, K Kavukcuoglu, D Silver, A Graves, I Antonoglou, D Wierstra, M Riedmiller, "*Playing Atari with Deep Reinforcement Learning*", DeepMind 2013

[5] Keras. `https://keras.io/`

[6] OpenGL. `https://www.opengl.org/`

[7] pyBullet. `https://pybullet.org`

[8] TensorFlow. `https://www.tensorflow.org/`

[9] Gym OpenAI. `https://gym.openai.com/`

[10] URDF XML Specifications. `http://wiki.ros.org/urdf/XML`

[11] Erwin Coumans. Bullet physics library. `http://bulletphysics.org/wordpress/`

[12] Shubhendu Trivedi, Risi Kondor *Lecture 6 "Optimization for Deep Neural Networks"*. University of Chicago,2017.

[13] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. "*Prioritized experience replay*" Google DeepMind.